Contextualizing Rename Decisions using Refactorings, Commit Messages, and Data Types

Anthony Peruma[a,*], Mohamed Wiem Mkaouer[a], Michael J. Decker[b], Christian D. Newman[a]

[a]*Rochester Institute of Technology, Rochester, NY, USA*
[b]*Bowling Green State University, Bowling Green, OH, USA*

Abstract

Identifier names are the atoms of program comprehension. Weak identifier names decrease developer productivity and degrade the performance of automated approaches that leverage identifier names in source code analysis; threatening many of the advantages which stand to be gained from advances in artificial intelligence and machine learning. Therefore, it is vital to support developers in naming and renaming identifiers. In this paper, we extend our prior work, which studies the primary method through which names evolve: rename refactorings. In our prior work, we contextualize rename changes by examining commit messages and other refactorings. In this extension, we further consider data type changes which co-occur with these renames, with a goal of understanding how data type changes influence the structure and semantics of renames. In the long term, the outcomes of this study will be used to support research into: 1) recommending when a rename should be applied, 2) recommending how to rename an identifier, and 3) developing a model that describes how developers mentally synergize names using domain and project knowledge. We provide insights into how our data can support rename recommendation and analysis in the future, and reflect on the significant challenges, highlighted by our study, for future research in recommending renames.

*Corresponding author
Email addresses: axp6201@rit.edu (Anthony Peruma), mwmvse@rit.edu (Mohamed Wiem Mkaouer), mdecke@bgsu.edu (Michael J. Decker), cnewman@se.rit.edu (Christian D. Newman)

Keywords: Program Comprehension, Identifier Names, Refactoring, Rename Refactoring, Data Types

1. Introduction

Program comprehension is the pillar of a developer's everyday development tasks; almost every programming task requires a certain degree of understanding of the existing codebase. In this way, software maintenance and evolution critically rely on the degree to which developers comprehend their codebases. Many studies demonstrate the significance of the effort, in terms of time, undertaken by developers when comprehending code [1, 2]. For instance, the time spent by developers in reading and comprehending code is significantly longer than the time spent in writing new instructions [2]. Therefore, developers' productivity can be optimized by decreasing the time needed for them to understand the existing code [3, 4, 5, 6].

One of the most atomic activities in source code development is the naming of code elements (e.g., class names, function/method names, etc.) collectively referred to as identifiers. Identifier names are the basic building blocks of program comprehension. Choices made when constructing identifier names directly impacts productivity [4, 6, 5, 7]. For example, abbreviated terms may hinder comprehension for both tools and humans. In response, studies search for ways to standardize and normalize identifier names to support both developers and tools [8, 9] and many research projects, both recent and otherwise, aim to enhance identifier naming using machine learning, static analysis, and by studying naming inconsistencies [10, 11, 12, 13, 14, 15].

One way to improve identifiers is to apply a rename refactoring [16]. Rename refactorings are defined as refactorings which modify the name of an identifier without modifying the intended behavior of the code for which the identifier is a part. Many Integrated Developer Environments (IDEs) offer a built-in rename refactoring functionality. Most of these IDEs only support the mechanical act of renaming; they allow a developer to choose what identifier they want to rename,

what new name should be used, and then perform checks to ensure that the new name will not introduce name collisions and that the new name is applied in all appropriate locations. There is little or no support to help inform developers of when to rename an identifier (e.g., when a name is of sub-optimal quality), and how to rename them (i.e., what words to use within the name). Instead, renames are typically performed when a developer notices that an identifier does not accurately reflect the behavior it represents. This causes renaming to be applied in a manner that is not always wholly systematic. Further, a developer is free to come up with whatever name they like (i.e., within the limits of naming conventions defined for the project). This new name may be even worse than the original, but there is no formal method to determine when this is the case.

Because naming heavily affects comprehension for both tools and humans, it is important to fully support developers when they must modify identifier names. That is, research must support developers in applying rename refactorings. Recent research on naming focuses heavily on suggesting identifier names [13, 14, 10, 11], studying how names correlate with behavior [12, 17], and analyzing names to reveal interesting properties [18, 19, 20, 21, 22]. We focus this work on investigating how names evolve [23, 24, 18, 25, 26] (i.e., are changed via rename) and how these changes affect/are affected by: 1) other changes made to the code (i.e., behavior preserving or not) as part of 2) a larger development plan/context. This information is critical if we are to support the evolution of identifier names through recommending when and how to rename an identifier.

In this paper, we study renames in two ways. 1) This paper utilizes a taxonomy of rename types published by Arnoudova et al. [18] to understand the types of changes applied to identifier names within our dataset. That is, we study how individual terms within an identifier are modified both syntactically and semantically when a rename refactoring is applied. 2) The paper contextualizes these rename types by analyzing changes to data types, commit log data, and refactorings which co-occur with renames. This allows us to understand how changes, to the code surrounding an identifier, affects changes to the identifier's name and, likewise, how development activities (i.e., written in a commit

log) affect changes to the identifier's name.

This paper is an extension of two prior works. Our initial work on renames investigated how method, class, and package identifier names evolve and how this evolution was described in commit messages. Our aim was to understand how names evolve and how this evolution is documented. Our assumption was that the choice of wording used during a rename is influenced by external factors which may appear in commit messages [23]. Through this work, we determined some preliminary trends in how development activities, such as adding features, influenced the terminology used during a rename. However, due to the limitations of natural language analysis techniques, and the occasionally information-light nature of commit messages, the results we found were not as actionable as we wanted. The trends we detected were nevertheless instructive and helped guide us toward ways to improve our approach. Thus, we extended [23] by taking into account the types of refactorings which occur before, or after, a rename refactoring while performing commit message analysis on these surrounding operations [24]. This allowed us to more clearly identify how names are influenced by their surrounding changes (e.g., *Extract Method* and *Move Attribute* frequently occur before certain types of renames) and how these influences are documented in commit messages. It also highlighted further challenges, which we discussed as research directions for ourselves and the larger software engineering research community.

In this paper, we extend our recent work [24] by additionally considering the situation where a rename is applied to an identifier, and that identifier's corresponding data type is changed. Data type changes are interesting because, unlike refactorings, they may change the external behavior of their associated identifier. Data types tell us what data and behavior an identifier represents (i.e., the data type tells us what attributes and methods can be used with this identifier). Therefore, when an identifier and its data type both change, this indicates a potential shift in behavior (e.g., added methods, new API), a shift in the data represented by an identifier (e.g., added attribute), or shift in the representation of data and behavior in a system (e.g., a change to improve

comprehension). By studying this situation, we can gain a more acute understanding of name evolution using a type of code change (i.e., data type changes) that has a stronger, direct influence on the behavior of a given identifier, and provide means for previous rename recommendation techniques to consider type migration as another dimension to learn a more suitable name.

The goal of this extension is to understand the influence that data type changes have on the structure and meaning of a rename. We emphasize data type changes for three reasons: 1) Changes to an identifier's type are relatively easy to detect in many programming languages. Therefore, making suggestions to developers on the fly when a type change is performed is already feasible in modern IDEs. 2) Types have strong influence over the data and behavior represented by an identifier, so changes to the type can have heavy significance on their associated identifier. 3) Type changes are a simple way for us to explore some non-refactoring code changes related to renames. These results will provide insight for our long term goals. In the long term, the outcomes of this study will be used to support research into: 1) recommending when a rename should be applied (e.g., after specific types of refactorings), 2) recommending how to rename an identifier (e.g., what words to use), and 3) developing a model that describes how developers mentally synergize names using domain and project knowledge. We provide insights into how our data can support future recommendations. Additionally, we expand our reflection on the significant challenges for future research in recommending renames. Hence, we answer the following research questions:

RQ1: What is the distribution of experience among developers that apply renames? We want to know how much experience developers who apply renames typically have. We use this question to understand the population from which our data has been obtained; contextualizing our data with respect to the level of experience of the developers it was generated by. This is important for future comparison with our dataset.

RQ2: What are the refactorings that occur more frequently with identifier renames? With this question, we aim to understand which types of

refactorings tend to occur before or after a rename. Our assumption is that the changes made to code immediately before or after a rename have a relationship with the rename itself.

RQ3: To what extent can we use refactoring occurrence and commit message analysis to understand why different semantic changes were applied during a rename operation? Using our refactoring co-occurrence data from RQ2, we add in commit message data in an effort to see how effectively we can pinpoint the development reason for certain changes (e.g., using more general words) to words in identifier names.

RQ4: What structural changes occur when an identifier and its corresponding type are changed together? When an identifier is renamed in-tandem with its data type, it may indicate a behavioral or semantic change since modifying the type ultimately may mean that the amount, or type, of data represented by an identifier has changed. This question explores structural changes made to an identifier to understand how type names are included in/removed from identifier names and changes to types affect structural changes made to identifier names.

RQ5: What semantic changes occur when an identifier and its corresponding type are changed together? When an identifier is renamed in-tandem with its data type, it may indicate a behavioral or semantic change since modifying the type ultimately may mean that the amount, or type, of data represented by an identifier has changed. We explore how identifier and type naming semantics evolve together in this question, including how the plurality of identifier names correlate with collection types and whether there is a covariant or contravariant relationship between semantic updates to type names and identifier names.

RQ6: What refactorings most frequently appear before and after an identifier and its corresponding type are changed together? Are there specific semantic changes which correlate with these refactorings? This question helps us understand if there is a common set of refactoring operations applied before or after renames which involve an identifier name and

its corresponding data-type. Further, we explore if these refactorings imply different semantic changes to the identifier name.

2. RELATED WORK

Since the choice of adequate naming for identifiers is critical for code understandability, there have been many studies that analyze the quality of identifiers and how identifier quality affects comprehension and developer efficiency. Hence, we divide our discussion of related work into two areas - studies that explore identifier renamings from a natural language perspective and studies that investigate the quality attributes that an identifier should exhibit.

2.1. Identifier Renaming

Arnoudova et al. [18] studies and proposes an approach to analyzing and classifying identifier renamings by mining rename operations and contrasting between the old and new forms of a given identifier. Their technique leverages the lexical database WordNet to understand how words in the old and new versions of an identifier are related. They additionally conduct a survey showing that 68% of developers want more support in naming identifiers and see name recommendation as an important problem. As part of their work, the authors introduce a taxonomy for identifier renames and report on the distribution of each type of classification, that was part of the taxonomy, in a dataset. Our work looks at understanding the motivations that drive developers to perform identifier renames and how these motivations affect the name's evolution. Hence, we utilize Arnoudova's taxonomy as a mechanism to measure the evolution of a name and contextualize this evolution using commit messages, refactorings, and data types. Additionally, our study examines a larger number of rename operation types compared with [18].

Our previous work [23] is a study on how terms in an identifier change, and we contextualize these changes by analyzing commit messages using a topic modeler; looking for words that indicate what development activity had occurred

with the change. We extended [24] to attempt contextualization using not only commit messages, but also the refactorings which surround a given rename (i.e., a refactoring is applied before or after the application of a rename). We found that certain refactorings, such as extract method, move attribute, move class, and rename (method, variable) were the most likely to occur before a rename and that these refactorings had motivations which could be obtained via the commit message, such as reverting the code (rename followed by rename) and architectural changes (move class). In this study, we further extend this work by considering rename operations which, in addition to changing an identifier's name, change the same identifier's corresponding data type; an important relationship to consider due to how the type is how developers reason about both behavior (e.g., methods) and data (e.g., attributes) that an identifier represents.

NATURALIZE, a framework, based on statistical language models, to mine natural source code naming conventions, was implemented by Allamanis et al. [27]. NATURALIZE looks for potential variable renaming opportunities by learning the coding conventions in the source code via syntactic restrictions, sub-grammars on existing identifier names. The authors extended this work [28] to suggest the renaming of methods based on their bodies and renaming classes based on their methods. Their work tries to find renaming opportunities, whereas we analyze renames to understand identifier name evolution. Our work focuses on analyzing how the words in a name evolve as a result of a rename and external factors, where external factors are obtained though commit message analysis or implied by other, co-occurring refactorings/data type changes.

Liu et al. [25] propose an approach that monitors the rename activities performed by developers and then recommends a batch of rename operations to all closely related code elements whose names are similar to that of the renamed element by the developer. They also study the relationship between argument and parameter names and utilize the patterns they found to detect naming anomalies and suggest renames to developers [26]. The authors determine what other names a developer should pay attention to given that a rename has been applied and regularity in the naming structure between parameters and arguments. The

focus of our work is complementary to this as we are looking at the semantics behind changes made to a name and the motivation for these changes, which could assist Liu's approach by helping determine how to rename an identifier (e.g., narrow the meaning) and provide more information as to when to rename (e.g., after a *Move Attribute*).

2.2. Identifier Name Quality

There are several recent approaches to appraising identifier names for variables, methods, and classes. Liu et al. [11] propose an automated approach based on deep learning to debug method names based on consistency between the method's name and its implementation. Kashiwabara et al. [14] use association rule mining to identify verbs that might be good candidates for use in method names; this work focuses on word co-occurrence to find any emergent relationships. [13] uses an ontology that models the word relationships within a piece of software. They then generate suggestions for new identifier names using different schemes for how to choose sequences of words to put together to form the identifier. Allamanis et al. [10] use a novel language model called the Subtoken Context Model, which is a neural network that has some similarity to n-grams (in that it uses a previously seen set of tokens to predict a new token). The difference is that the neural network is able to take into account long-distance features (e.g., identifier names that occur very far away from the target location) and produce neologisms (essentially, new identifiers) by concatenating words together (i.e., as is commonly done by developers).

Liblit et al. [29] discuss naming in several programming languages and make observations about how natural language influences the use of words in these languages. Schankin et al. [4] focus on investigating the impact of more informative identifiers on code comprehension. Their findings show an advantage of descriptive identifiers over non-descriptive ones. Hofmeister et al. [5] compare comprehension of identifiers containing words against identifiers containing letters and/or abbreviations. Their results show that when identifiers contained only words instead of abbreviations or letters, developer comprehension speed

increased by 19% on average. Lawrie et al. [6] study 100+ programmers, asking them to describe twelve different functions. These functions used three different "levels" of identifiers: single letters, abbreviations, and full words. The results show that full word identifiers lead to the best comprehension, though there were cases where there was no statistical difference between full words and abbreviations. Butler et al. [7] extend their previous work on java class identifiers [30] to show that flawed method identifiers are also (i.e., along with class identifiers) associated with low-quality code according to static analysis-based metrics.

Høst and Østvold [12] designed automated naming rules using method signature elements, i.e., return type, parameter names, and types, and control flow. They call this technique method phrase refinement, which takes a sequence of part of speech tags (i.e., phrases) and concretizes them by substituting real words. (e.g., the phrase <verb>-<adjective> might refine to is-empty). Additionally, they use static analysis to group method names (in phrase form) together by behavior. Binkley et al. [21] presented empirically-derived rules that certain types of identifiers (e.g., class field identifiers) should follow. One of these rules is that class fields should never be just an adjective.

Arnaoudova et al. [17] define a catalog of linguistic anti-patterns that are found to deteriorate the quality of code understanding. The authors show the negative impact of linguistic anti-patterns by conducting two studies with software developers and finding that the majority of programmers perceive anti-patterns as poor naming practices. In their study of readability metrics, Fakhoury et al. [31] show that current metrics may not be effective at capturing readability improvements; highlighting the importance of further research into the quality of naming and how names evolve over time.

Previous studies consider the current state of the software to analyze identifiers, without considering the history of their previous renames. We complement these studies by examining how names evolve, the semantics behind this evolution, and correlated code change activities that motivate this evolution. While existing studies focus on static names (i.e., identifiers in their current state rather than analyzing how they change), our approach focuses on how names

evolve over time. Thus, we add to the body of knowledge represented by these related papers. The dataset we collected can be used to augment the accuracy of rename recommender techniques, along with unlocking the possibility to perform empirical studies and understand to what extent developers follow naming conventions and best practices over a system's lifetime.

3. ANALYSIS OF RENAMES

The hypothesis of this paper is as follows: Changes to the name of an identifier are most likely related to other code changes made locally (i.e., in the same class, function, or file) and/or the motivation behind those changes (e.g., using a new API, fixing a bug). Under this hypothesis, we should be able to correlate the types of changes made to a name with other local code changes and/or change descriptions (e.g., in commit messages). We use a taxonomy created by Arnaoudova et al. [18] to analyze rename refactorings and categorize them into the different types prescribed by this taxonomy. In this section, we briefly discuss the taxonomy, but encourage the reader to read the original work for a more thorough discussion of each category. We also take a look at examples of renames and how they are influenced by surrounding changes found in our dataset. We will start with the taxonomy.

3.1. Taxonomy for Rename Refactorings

Entity Kind: Entity kind records the source code entity that a given identifier represents. For example, the identifier may be the name of a type, class, getter, setter, etc.

Form of Renaming: This category reflects the lexical change made to the identifier. It is broken down into a few subcategories: Simple, Complex, Reordering, and Formatting. Simple changes are those that only add, remove, or change one term in the identifier. Complex changes add, remove or change multiple terms. Reordering is where two or more terms in an identifier switch positions (i.e., GetSetter becomes SetterGet), and Formatting changes are those

where there is no renaming but a letter in a term changes case or a separator (e.g., underscore) is added or removed.

Semantic Changes: These are changes due to adding/removing terms or modifying terms (e.g., to another term that is a synonym of the original) such that the meaning of the identifier may have been modified. The following heuristics are used to figure out whether the idenifier's semantics have been preserved or modified.

We consider the identifier's **meaning preserved** if one of the following holds: 1) the change added/removed a separator, 2) the change expanded an abbreviation, 3) the change collapsed a term into an abbreviation, 4) the old term was changed to a new term which is a synonym of the old term, 5) multiple old terms were changed to multiple new terms which are synonyms OR use or removal of negation preserves meaning of the identifier (i.e., ItemNotVisible becomes ItemHidden).

We consider the identifier's **meaning modified** if one of the following holds: 1) *Broaden meaning*– the old term is renamed to a hypernym of itself OR a term (i.e., adjective or noun) was removed which generalizes the identifier (e.g., GetFirstUnit becomes GetUnit). 2) *Narrowing meaning*– the old term is renamed to a hyponym of itself OR a term was removed which narrows the meaning of the identifier (e.g., GetUnit becomes GetFirstUnit). 3) We consider *meaning changed* (i.e., not narrowed or broadened) when an old term is changed to a new term which is unrelated to the old; when a new term is the old term's meronym/holonym, or antonym; OR when multiple terms are changed AND a negation reverses a synonym of the old term. 4) *Add meaning*– one or more new terms were added to the identifier AND the addition does not fall into one of the categories above (e.g., narrow meaning). 5) *Remove meaning*– one or more terms removed from the identifier AND the removal does not fall into one of the categories above (e.g., broaden meaning).

3.2. Contextualizing Rename Refactorings

Developers rename identifiers for multiple reasons. Through careful analysis of rename refactorings, one can gain insight into how developers choose their words, why they choose certain types of words over others, and how to mimic this process automatically. In this subsection, we show examples of how developer activity, recorded in commit messages and refactoring operations, is reflected in their renaming choices.

By analyzing the following method rename: *setDisableBinLogCache* → *setEnableReplicationCache*, we observe that the meaning of the name has changed; the developer has modified the name by changing *disable* to *enable*. This change is reflected in the commit message entered by the developer: *"Changes replication caching to be disabled by default"* [32]. Similarly, the renaming of a class from *Key* → *EntityKey* demonstrates an act of narrowing the meaning of the identifier. Once again, the purpose of this rename is reflected in the commit message: *"Rename Key to EntityKey to prepare specialized caches"* [33].

Developers may also rename identifiers to: 1) better represent the existing functionality and not when they are changing or narrowing it, or 2) adhere to naming standards or correcting a spelling/grammatical mistake. For example, here the developer renamed the class *TestProxyController* → *ProxyControllerTest* by reordering the term names to *"...fixed names that were not in standards"* [34]. In the next example, the developer preserves the meaning of a method by renaming it from *inactivate* → *deactivate*, through the use of a synonym. This is, again, reflected in the commit message: *"Renaming method to proper English..."* [35], where renaming to 'proper English' indicates that the meaning has not been modified but should now be easier to comprehend.

Finally, commit messages are not the only way to contextualize rename refactorings. Changes to the code surrounding a name also help in understanding what the developer's intention. Unfortunately, most types of changes to the code are not part of a pre-defined taxonomy. That is, it is difficult to understand the abstract, domain-level goal of individual changes. Luckily, some types

of code changes are taxonomized. Specifically, refactorings are a taxonomy of changes made to the code for a specific goal; typically to optimize non-functional attributes of the code [16]. We can look at refactorings that happen just before and right after a given rename to help us understand what the developer was doing before and after they applied a rename refactoring.

For example, in commit [36] the developers applied an *Extract Method* refactoring with the following comment: "using the Jangaroo parsing infrastructure; all tests green; getters inherited", before applying rename: *getCompilationsUnit* → *getCompilationUnit*. This preserves the meaning of the name but puts the name more in-line with its type, as stated by the commit message for this change: "Corrected type in internal method name" [37].

Another example comes from a move class refactoring, where a class was moved from one package to another [38]. This refactoring commit had the following comment: "Incremental changes, some package refactorings etc". Further, a rename was performed after this commit: *JsonViewResult*→*JsonView* [39]. This rename broadens the meaning of the name by removing *result*, making the identifier more general in meaning. The commit message associated with the rename is: "Cleaned up some file names for easier usage...", meaning the developer was likely going through and renaming things after the move class refactoring.

In addition to surrounding code changes, a change in the data type associated with an identifier can also help contextualize a rename of an identifier. For example, in commit [40], the developer performs the following *Rename Variable*: *Date sqlDate* → *Timestamp timestamp* with the commit message "fixes issue #29. java.util.Date and jodatime.Datetime instances would loose time information..." From this example, we see that reason for the rename is to fix a bug by utilizing the `Timestamp` data structure instead of `Date`.

The question we ask, in the context of these examples, is whether there are overarching themes to the way names change given that a refactoring or data type change has occurred in a commit surrounding it. If so, then it is possible to study these trends and use them to support developers in their naming activities.

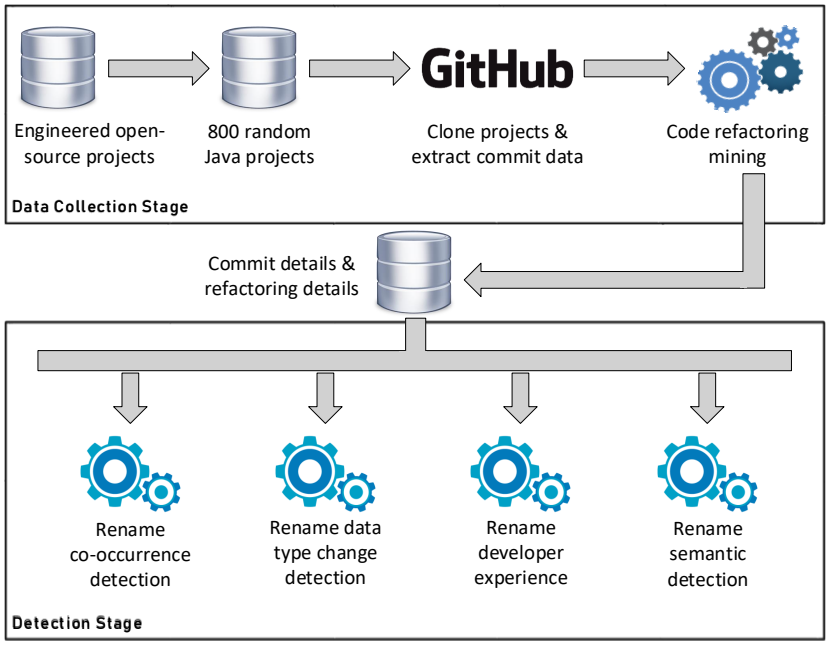

Figure 1: Methodology overview

4. Methodology

Our methodology consists of two stages - Data Collection and Detection. The Data Collection stage consists of constructing our dataset while the Detection stage consists of examining and querying the dataset for specific characteristics to help answer our research questions. Figure 1 represents an overview of the approach used to conduct our experiments. In the subsequent subsections, we explain in detail the approach for each activity. Furthermore, the dataset utilized in this study is available on our project website [41]. Due to performance requirements associated with this volume of data mining and data analysis, the activities associated with both phases were performed on a dedicated virtual machine with 16 GB of RAM, and a 3.40 GHz i7 CPU. With this configuration, the Data Collection Stage took approximately four weeks to complete, while the Detection Stage was completed in around 1.5 weeks.

4.1. Data Collection Stage

Projects: A key element to an empirical research study is the relevance of the dataset on which the study is based. To obtain a viable dataset, we select 800 random, open-source Java projects hosted on GitHub. These projects are part of a curated dataset of engineered software projects made available by [42]. The authors of this dataset classified engineered software projects based on the project's use of software engineering practices such as documentation, testing, and project management. For each of these 800 projects, after cloning the project repository, we enumerate over the commit log of each project to extract metadata associated with each commit. The extracted data includes the author (name and email) who was responsible for the original creation of the commit, the creation timestamp of the commit, and the names of the files that were part of the commit.

Refactorings: To obtain the set of refactorings from each project, we utilize RefactoringMiner [43]. At the time of our study, RefactoringMiner can detect 28 different refactoring operations. From this list of operations, seven are rename based operations. At a high-level, we utilize RefactoringMiner to iterate over all commits of a repository in chronological order. During each iteration, RefactoringMiner compares the changes made to Java source code files in order to detect refactorings in the code based on a pre-defined set of refactoring rules. While there are a few other tools that can mine refactoring operations [44], we selected RefactoringMiner since it represents state of the art in the field of refactoring detection [45], along with a precision of 98% and a recall of 87% [43, 46]. Therefore, it is well suited for our large-scale mining study. We investigate the renaming operations on five types of identifiers - classes, attributes (i.e., class-level variables), methods (including getter and setters), method parameters, and method variables. Furthermore, we conduct our experiments on the entire commit history of the project (and not on a release-by-release comparison).

4.2. Detection Stage

Rename Forms & Semantics: We utilize the tool from one of our prior studies [23] for the detection of rename-based form and semantic updates made to an identifier's name. The tool follows the rules specified by Arnaoudova et al. [18] to determine the type of form and semantic change an identifier name undergoes when renamed. Input for the tool is the pair of old and new names associated with a renamed identifier.

First, from the output provided by RefactoringMiner, we extract all rename-based refactoring operations. Next, from these operations, we extract: 1) each pair of old and new names, 2) the name of the source code file containing the renamed identifier, 3) the name of the class containing the renamed identifier, and 4) the unique ID of the commit associated with the refactoring.

Since most identifier names are composed of multiple terms, a pre-requisite to performing the form and semantic analysis is the splitting of each name into its constituent terms. Hence, the tool utilizes the Ronin splitter algorithm implemented in the Spiral package [47] to determine the terms that form a name. The tool primarily relies on Python's Natural Language Toolkit (NLTK) [48] to compare the old and new identifier name to determine the type of semantic change made by the developer. To determine the relationship between terms in the names, the tool makes use of WordNet [49], to obtain the semantic and part of speech details about each term.

Renames With Data Type Changes: We built a custom tool to identify data types associated with identifiers that undergo a rename. Based on Java technical documentation [50], our experiments consider the following eight data types as primitives: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. Additionally, we examine the distribution of data types that store a group of values/references (i.e., arrays and collections) [51]. For methods, we consider the return type of the method as the data type. As `void` is not considered a type in Java [52], we exclude instances where the return type changed to/from `void`. This exclusion allowed our analysis on methods to be consistent with

the other identifiers that have types– attributes, method variables, and method properties; these identifiers must be associated with a data type (either primitive or non-primitive) and thus cannot have void as their type. However, for all other instances, we apply the same processing we performed on the other identifiers in our experiment.

Our study of data type changes and their involvement in rename refactorings is limited to attributes, methods, method parameters, and method variables since classes do not have types. For each rename instance of these types of identifiers, we extract the name of the data type associated with the old and new name of the identifier. For example, the *Rename Attribute* refactoring `long connTimeToLive` → `TimeValue timeToLive` also contains a change in data type. In this instance, the developer changes the type of the identifier from `long` to `TimeValue` when renaming the attribute from `connTimeToLive` to `timeToLive`.

Rename Co-occurrence With Refactorings: We built a custom tool to identify refactorings that occur before and after rename refactoring. The tool functions by iterating over the commits which contain refactorings in our dataset. This is done in chronological order (based on the author-date – the date the commit was originally made). Since our rename refactorings are related to classes, attributes, methods, method parameters, and method variables, we restrict our detection to refactorings that are applied to only these types of identifiers. For each renamed identifier type, we first extract all unique instances. Next, we iterated through all refactorings searching for refactorings that involved the specific instance. Our process does not take into account the time duration between commits when looking for surrounding refactoring commits.

To better highlight this process, consider the example where we detect the class `stormpot.CountingAllocatorWrapper` as being renamed to `stormpot.CountingAllocator` [53]. We first query our list of unique attributes, methods, parameters, and variables for identifiers that were part of this class and had also undergone a refactoring. Our search results in an attribute, `counter`,

belonging to this class, which had undergone a rename refactoring (prior to the class being renamed) [54]. We utilize the author-date attribute associated with a commit to determine the order of the commits. Finally, we record this pair of refactorings in our database. It should be noted that the version of RefacotringMiner we utilize only supports rename refactoring operations for parameters. Hence, we did not obtain other types of refactoring operations that developers might apply to parameters.

Rename Co-occurrence With Data Type: The purpose of this activity is to detect and analyze the co-occurrence of rename refactorings that also contain a data type change to the renamed identifier. Hence, we follow an approach similar to the general rename refactoring co-occurrences described above. However, in this new approach, we limit the dataset to only rename instances with a data type change; the general approach did not consider data type changes. For example, the *Rename Attribute* refactoring `HistoryMap historyMap` → `History history` contains a data type change from `HistoryMap` to `History` [55]. However, before performing this rename, the developer performs a *Pull Up Attribute* refactoring operation on the attribute [56]. In this instance, the refactoring operations *Rename Attribute* and *Pull Up Attribute* co-occur when the data type of the attribute changes during its rename.

Commit Log Analysis: To derive the developer's rationale for performing a rename, we look at the commit log as a means of contextualizing the rename. Hence, our experiment involves the performance of a topic modeling and n-gram analysis of commit messages. For our topic modeling analysis, we utilize the Latent Dirichlet Allocation (LDA) [57] algorithm. Additionally, we use a combination of topic coherence [58] and manual empirical analysis as a means to determine the ideal number of topics; past research has shown that the number of topics can vary between studies and datasets [59]. A prerequisite to these activities was a text preprocessing task where we cleansed and normalized the commit messages. Normalization is a process of transforming non-standard words into a standard and convenient format [60]. Some key steps in our pre-

processing include: removal of stopwords, URLs, numeric and alphanumeric characters/words, and non-dictionary words. Additionally, we also expand contractions (e.g., 'I'm' → 'I am') and perform stemming and lemmatizing on words.

Taxonomy: To support our discussion of the challenges involved in our study (refer Sections 6.3 and 6.2), and to better understand the rationale behind the renaming of an identifier, we perform a qualitative analysis on the source code changes that accompany rename refactorings. In this experiment, we manually review the diff of the commit in order to understand if the rename was made in conjunction with other changes to the code or by itself. As a setup for this experiment, we select 30 random rename instances from each of the five types of identifiers. This results in a total of 150 source code files for our manual review; where at least two authors review each file. When performing the review, the reviewers first examine changes made by the developer to surrounding code elements and the commit message. Next, the reviewers determine the rationale for the rename. Finally, the reviewers compare their individual taxonomy annotations and agree on a final set. The reviewed source files were then annotated using this finalized taxonomy.

Developer Experience: The purpose of this activity is to determine the experience of the developers that refactor the source code in a project. As our study is on renames, we derive the experience of developers where the developers refactoring operations are limited to only renames, developers who perform all refactoring operations, and developers who perform only non-rename refactoring operations. As this is a large empirical study, obtaining the experience of each developer, associated with a project, in our dataset is not feasible and can also be subjective. Hence, to overcome this challenge, we perform a more objective-based experiment where we follow the approach utilized by [61]. In their approach, the authors use project contribution as a proxy for developer experience within a project. Hence, for each developer in each project, we calculate the Developer's Commit Ratio (DCR). This ratio measures the number of

individual commits made by the developer against all project commits. In other words, DCR = $(\frac{IndividualContributorCommits}{TotalAppCommits})$. We utilize the project's commit log along with the output of RefactoringMiner to determine the developers that belong to each of the three groups. Using details in the commit log, we first calculate the DCR for all developers in a project. Next, using the output of RefactoringMiner, we split the developers into their respective groups based on the type of refactoring operations they had performed during the lifetime of the project. To mitigate the threat of misattributing commits due to the use of GitHub features such as pull requests, we only consider the author of a commit as its developer.

5. Experimental Results

In this section, we discuss the results of our experiments. The discussion is broken down into six Research Questions (RQs). While RQs 1-3 focus on all rename refactorings, RQs 4-6 focus specifically on rename refactorings in which the renamed identifiers also had a change in data type. The RQs are designed to help us understand how data type changes affect the evolution of identifier names when these changes are applied in tandem. In RQ1, we focus on the experience of developers that perform rename refactorings versus other types of refactoring operations. In RQ2, we discuss what types of refactorings occur before or after a rename refactoring. Additionally, we look at how often rename refactorings are preceded or followed by another refactoring, and what types of refactorings these preceding or following changes represent. In RQ3, intending to contextualize identifier renames, we combine and discuss data from RQ2 with commit message information and the semantic change types discussed in Section 3.1. Our end goal is to utilize the commit message and refactoring information to contextualize the semantic change types we detected in our set of renames. In RQ4, we examine the structural changes applied to an identifier name when both it and its corresponding type are changed together. In RQ5, we apply similar analysis as in RQ4 except we look at semantic, instead of

Table 1: Distribution of the top five refactorings

Refactoring Type	**Count**	**Percentage**
Rename Attribute	137,842	19.37%
Rename Variable	84,010	11.81%
Rename Method	82,206	11.55%
Move Class	76,265	10.72%
Extract Method	47,477	6.67%
Others	*283,695*	*39.87%*

structural, changes; identifying how the meaning of identifier names evolve when their type is changed in-tandem. Finally, RQ6 is similar to RQ2, but we focus on refactorings surrounding identifier renames which include a change to the corresponding type.

5.1. Data Summary

For context, we present a summary of our dataset before we discuss our results. First, with regards to project cloning, in total, we collected 748,001 commits with a project containing 732 commits and 19 developers on average. In terms of recentness, the projects were cloned in early 2019, and approximately 74.6% of the projects had their most recent commit within the last four years. Next, looking at the RefactoringMiner output, we identified 711,495 refactoring operations, with each project in our dataset exhibiting more than one refactoring operation. After the removal of outliers (via the Tukey's fences approach), on average, each project had 450.8 refactoring operations performed by seven developers. Approximately 53.51% of the refactoring operations in our dataset were rename based. We present the top five refactoring operations, from our mined dataset, in Table 1.

Looking at the form type and semantic updates data, obtained during the Detection stage (Section 4), we observed that developers more frequently perform a Simple form type rename compared to Complex, Formatting, and Re-

Table 2: Distribution of rename forms and semantic meaning updates made to identifier names by developers

Type	Count	Percentage
Rename form types		
Simple	259,754	68.31%
Complex	109,860	28.89%
Formatting	8,916	2.34%
Reordering	1,732	0.46%
Rename semantic meaning updates		
Preserve	29,568	7.78%
Change	350,694	92.22%
Change – Narrow		44.21%
Change – Add		37.93%
Change – Broaden		15.09%
Change – Remove		2.58%
Change – Antonym		0.19%

ordering. In terms of semantic updates, most identifiers undergo a change in meaning, with a narrowing in meaning occurring the most. Shown in Table 2 is the distribution of rename form and semantic meaning types that were performed by all developers in our dataset.

From our analysis of renames with data type changes, approximately 17.39% (53,962) of renames were performed with a change in data type. From this set, developers frequently change the type of method variables followed by method parameters. A breakdown into the individual identifier types is presented in Table 3. Out of the 800 projects in our dataset, 769 ($\approx 96.13\%$) of these projects exhibited rename instances that had a change in data type. Looking at the indi-

Table 3: Distribution of rename-based type changes

Type Changed	Type of Rename	Count	Percentage
No	Rename Attribute	128,486	41.41%
No	Rename Variable	61,665	19.87%
No	Rename Method	37,923	12.22%
No	Rename Parameter	28,086	9.05%
Yes	Rename Variable	21,885	7.05%
Yes	Rename Parameter	16,285	5.25%
Yes	Rename Attribute	9,355	3.01%
Yes	Rename Method	6,397	2.06%
No	Move And Rename Attribute	187	0.06%
Yes	Move And Rename Attribute	40	0.01%

vidual identifier types, approximately 80.25% of all projects in the dataset had an attribute rename with a change in data type, while approximately 73.65%, 92.25%, and 81% of projects had a rename of a method, variable, and parameter occurring in tandem with a data type change respectively. Furthermore, approximately 42.75% of the projects from our dataset of 800 contained a refactoring occurring either before and/or after a rename refactoring that also contained a data type change.

Finally, we followed [18]'s approach to identify documented renamings in our dataset. From the set of mined rename refactoring commits, approximately 6.9% (or 4,701 out of 68,121) of the commits documented the renaming, compared to less than 1% in [18]'s dataset. This means that most commit messages do not explicitly discuss the rename operation. However, while renames are not always documented, the motivation behind the rename may still be gleaned from the commit message (e.g., the commit may discuss clean-up, bug fixing, changing a method's behavior). Not all commit messages which document renames specify why the rename is needed (e.g., "renaming some variables" [62])

and, likewise, some rename motivations can be found in commit messages which do not mention the rename itself (e.g., "extract method to convert db entity to generic entity" [63]). This percentage does indicate a potential need for rename documentation support.

We have made available, on our website [41], the dataset utilized in this study for replication and extension purposes.

5.2. RQ1: What is the distribution of experience among developers that apply renames?

To compare the distributions of DCR for developers who had performed only renames, only non-renames, and a mix of rename and non-rename refactorings, we follow the same approach as [61]. Since the number of developers in each project differs, we calculate an adjusted DCR value for each developer by dividing the developer's original DCR value by the number of developers in the project. We also restrict our experiment to projects that had only two or more developers. Figure 2 depicts the distribution of DCR values for developers based on the type of refactoring performed in their project.

Our observation of developers who perform all types of refactorings having a higher DCR than those that perform only rename refactorings is in line with the research indicating that rename operations are considered simpler, or more accessible, compared to other refactoring operations. That is, developers who are less experienced feel more comfortable applying them [64, 65, 66]. However, it is interesting that developers who perform only renames share a similar DCR value as those that perform only non-rename refactorings. To further validate these findings, we perform a nonparametric Mann-Whitney-Wilcoxon test on the DCR values for developers that belonged to these categories. We obtained a statistically significant p-value (< 0.05) when the DCR values of developers who performed only rename refactorings were compared to developers that perform all types of refactorings. This value confirms that developers that contribute less to a project are more likely to perform rename refactorings, which are generally considered easier to apply due to wide IDE support despite developers also

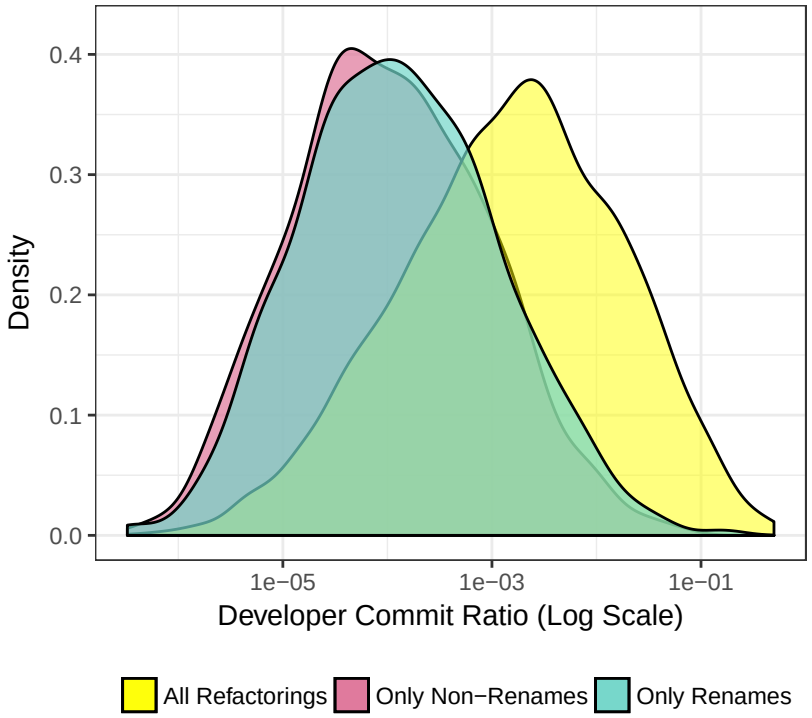

Figure 2: Distribution of DCR values for developers based on the type of refactoring performed in their project

generally agreeing that renaming is a difficult problem [18].

Looking at the different types of identifier rename forms, we observe that there is no significant difference in the distribution of renaming forms between developers that perform only renames and those that perform all types of refactorings. Similarly, the types of semantic updates to an identifier name also showed no significant differences among these two groups of developers. Table 4 provides a breakdown of the distribution of rename form and semantic meaning updates based on developer type. Our experiment on developer experience shows that developers with more project experience (i.e., contributions) are more accustomed to performing a multitude of different types of refactoring operations. This is not surprising as these developers have more experience

and knowledge of the codebase (and system) and would be more comfortable in implementing design/structural changes to the project. Given that rename refactorings have broad IDE support and are syntactically simple modifications, inexperienced developers will naturally be drawn into making such refactorings in the project.

Summary for RQ1: Developers with limited project experience are more inclined to perform only rename refactorings than other types of refactorings (which may alter the design of the system). This is an important context for any future recommendation effort and particularly for our data. Given that many of the developers performing the renames we analyzed have less experience on average, our results may reflect this lack of experience. Further research is needed to confirm the connection between the quality, of renames and developer experience.

5.3. RQ2: What are the refactorings that occur more frequently with identifier renames?

To derive the extent to which non-rename refactorings can either influence or be influenced by a rename, we study *the type of refactoring commits that occur just before and after a rename refactoring commit.* This is based on the idea that renames are likely to occur with other refactorings; an assumption supported by Murphy-Hill et al. [67] who shows that developers perform renames in batches more so than other refactorings and, most often, that refactorings occur on multiple related code elements. This part of our study focuses on the renames of classes, attributes, methods, method parameters, and method variables. For each entity type, we extract the list of unique instances that underwent a rename and then search for the refactoring that directly precedes and directly follows (i.e., there may be non-refactoring commits that we skip) the rename for either the same entity or child entities (as in the case of classes and methods).

Interestingly, we observe that for all elements that are subject to renames, developers frequently perform the rename in isolation with respect to other refactorings. In other words, approximately 91.97% (or 349,731) of rename

Table 4: Distribution of rename form and semantic meaning updates split by developers who performed all refactoring operations and those that performed only rename refactoring operations.

Type	**Only Renames**	**All Refactorings**
	Percentage	**Percentage**
Rename form types		
Simple	64.65%	67.01%
Complex	30.55%	29.96%
Formatting	4.56%	2.52%
Reordering	0.24%	0.51%
Rename semantic meaning updates		
Preserve	9.97%	8.50%
Change	90.03%	91.50%
Change – Narrow	48.99%	48.08%
Change – Add	29.93%	32.68%
Change – Broaden	18.33%	16.46%
Change – Remove	2.58%	2.56%
Change – Antonym	0.17%	0.21%

Table 5: Top 3 refactoring operations that occur before a class, attribute, method and method variable are renamed

Refactoring Operation	**Count**	**Percentage**	**Commit Message Key Terms**
Refactoring operations before a class rename			
Move Class	3,069	26.96%	package, structure, change
Rename Method	2,062	18.12%	code, clean, change, fix
Rename Variable	1,376	12.09%	add, code, test, support
Others	*4,875*	*42.83%*	*N/A*
Refactoring operations before an attribute rename			
Move Attribute	1,499	83.32%	added, fix, support, test
Pull Up Attribute	220	12.23%	added, simplification, extract
Push Down Attribute	73	4.06%	separate, remove, added
Others	*7*	*0.39%*	*N/A*
Refactoring operations before a method rename			
Rename Method	1,760	19.58%	revert, implementation, test
Extract Method	1,666	18.53%	fix, added, modified, test
Rename Variable	1,364	15.17%	added, test, fix, change
Others	*4,201*	*46.72%*	*N/A*
Refactoring operations before a method variable rename			
Rename Variable	3,067	90.66%	revert, added, test, fix
Extract Variable	305	9.02%	added, string, test, fix
Inline Variable	6	0.18%	fix, working, change
Others	*5*	*0.15%*	*N/A*

commits had no refactorings occur one commit before or one commit after. However, this does not mean that rename is the only action applied to this element during its lifetime. Upon the inspection of some cases, there were changes, applied to the element, which are not considered refactoring (e.g., adding lines of code to a method, adding a given identifier as a parameter to another method). For scenarios where there are refactorings either before or after a rename, we noticed that more operations occur before a rename ($\approx$ 6.27%) than after ($\approx$ 1.73%).

In general, the majority of the refactorings that occur before a rename are related to changes/updates to functionality. Additionally, we observe that some of these commits are bug fix related or due to developers either adding or updating unit test files. For example, in order to include new functionality, a developer refactors the existing code by creating a new method called `getClassURL` by performing an *Extract Method* operation [68]. Thereafter the developer renames the newly created method to `getClassUrl` to ensure that name follows "Google's style rules" [69].

Even though the number of refactorings occurring after a rename is much smaller, we did notice that most of these refactorings are associated with some form of code reversal/reverting. As an example, a developer initially renames a method from `getIncludedPublishers` to `getEnabledSources` when introducing new functionality [70]. However, in a subsequent commit [71], the developer removes this functionality from the method and also reverts back to the original method name.

As the majority of refactoring operations occur before a rename, in the following subsections, we drill-down into each element type with the aim of discovering the common types of refactorings that precede the renaming of the element and also the extent to which the commit log can contextualize the relationship between these refactorings. Table 5 highlights the distribution of the top three refactoring operations that occur before a class, attribute, method, and method variable is renamed. Also provided in this table are the common terms we extracted from our topic-modeling and n-gram analysis of the commit

messages that are associated with these refactoring operations. The complete list of refactorings that proceed and follow a rename refactoring is available on our project website.

5.3.1. Class Rename

Our study of class renames involve identifying the refactorings performed on the class and all elements within the class (i.e., attribute, methods, method parameters, and method variables) immediately before and after the developer renames the class. We observe that developers more frequently performed a *Move Class* refactoring before renaming the class. Results from our topic modeling and n-gram analysis coupled with a manual analysis of random messages showed that activities related to restructuring project structures and change of package names cause developers to rename class names. For example, in [72] a developer moves the class `BasicAuthLoginCommand` from `com.heroku.api.command` to `com.heroku.api.command.login` with the message "reorganized commands into appropriate packages." The next refactoring operation [73] performed on this class is renaming the class to `BasicAuthLogin`. The reason for the rename is "...to simplify some of the names."

Looking at the number of non-refactoring commits that separate a *Move Class* from a *Rename Class*, we observe that the majority of renames ($\approx 7.15\%$) occur in the commit immediately following the move. We also observe that a majority of these pairs of refactoring commits fell within one to five commits of each other – approximately 27.73% of the time. This lends support to the idea that they are related; *Move Class* refactorings are frequently done near the same time as *Rename Class*. While further investigation is required to determine when it is appropriate to recommend a rename in this situation, our data highlights this relationship as a good avenue for future, deeper research into what indicates that the rename will be performed versus when it will not be.

5.3.2. Attribute Rename

Similar to classes, developers perform move operations on attributes before renaming them. Looking at the commit messages, change in functionality (specifically adding of new features) is one of the most common reasons developers move an attribute. As an example, in commit [74], the developer moves the attribute `String jobId` with the message "added the jobId to a few more logs". The subsequent refactoring commit [75] for this attribute involves a renaming operation in which the attribute is renamed to `context` as part of a "cleanup" activity. We observe that around 71% of the renames occur in the commit immediately after the developer moves the attribute. Additionally, around 82% of rename refactorings take place within five commits after the *Move Attribute* operation.

5.3.3. Method Rename

For methods, we investigate the refactorings that are applied to the method and its members (i.e., parameters and variables) just prior to and after the method is renamed. Interestingly, we observe that developers perform a rename to the method before renaming it again more than any other type of refactoring. Based on the terms in the commit log, we observe that the reason for the initial rename is due to developers changing the behavior/purpose of the method. Furthermore, we notice that the second occurrence of the method rename reverts the first rename operation. For example, in [76], the developer renames the method `showDelivery` to `showOwnDelivery` as part of a functionality change, with the commit message "Minor changes to access controls in instructor MVC". In the subsequent commit [77], the developer reverts the name change as part of cleanup activities with the message "Final tidy of older instructor MVC".

Looking at the interval between commits, the majority ($\approx 15.22\%$) of the method-rename pairs of refactorings occur one after another. Further, a gap of between 1 to 5 commits occurs around 37.68% of the time between two method renames.

5.3.4. Method Variable Rename

Like methods, method variables also undergo rename operations in succession. Once again, looking at the commit messages, we observe that the reason for the initial rename tends to be due to either refactoring or change (including reversals) in functionality. It is also interesting to note that the developers revert the variable name of the initial commit in the next rename. For example, in [78] the developer renames the variable `drop` to `assembledDrop` with the message "simplified drop assembly a bit". The next commit [79] reverts the variable name when the developer performs a "misc code cleanup" activity. Finally, a gap of between 1 to 5 commits occurs around 33.94% of the time between two variable renames.

Summary for RQ2: We show that in most scenarios, renaming of an element does not generally seem to be influenced by, nor does itself influence another type of refactoring on the same element. This indicates that an analysis of non-refactoring operations will be required to understand how changes to code around a rename affect or are affected by the rename. However, there is a subset of renames that occur directly before or after another refactoring. Most commonly, the refactorings occurring before a rename are *Extract Method*, *Move Attribute*, *Move Class*, and *Rename*. In particular, we observe that 71% of the time, a rename occurs in the commit directly following a *Move Attribute*, and 82% of the time, this rename is within five commits after the *Move Attribute* operation. In other cases (*Move Class*, *Rename Method*), this percentage rests between 15 and 27%. Finally, in situations where a rename follows another rename, we observe that developers revert to the original name when performing the second rename.

5.4. RQ3: To what extent can we use refactoring occurrence and commit message analysis to understand why different semantic changes were applied during a rename operation?

To answer this question, we look at the types of semantic changes applied to identifier names given that another refactoring was applied in the previous

Table 6: An overview of the types of semantic updates an identifier name undergoes

Identifier Type	**Refactoring Before Rename**	**Top 3 Types of Rename Forms**	**Type of Semantic Update**	**Top 3 Semantic Change Subtypes**
Class	Move Class (Total Count: 3,160)	Simple (57.82%) Complex (34.68%) Formatting (5.54%)	Change (84.14%) Preserve (15.85%)	Narrow (63.56%) Broaden (28.13%) Add (3.65%)
	Rename Method (Total Count: 2,179)	Simple (65.26%) Complex (30.29%) Formatting (2.98%)	Change (90.0%) Preserve (10.0%)	Narrow (57.42%) Broaden (31.56%) Add (6.78%)
	Rename Variable (Total Count: 1,479)	Simple (61.19%) Complex (34.69%) Formatting (2.64%)	Change (100.0%)	Narrow (100%)
Attribute	Move Attribute (Total Count: 1,499)	Simple (67.44%) Complex (29.75%) Formatting (2.54%)	Change (94.66%) Preserve (5.34%)	Add (54.05%) Narrow (24.59%) Broaden (16.07%)
	Pull Up Attribute (Total Count: 220)	Simple (55.91%) Complex (35%) Formatting (8.18%)	Change (85.0%) Preserve (15.0%)	Narrow (66.84%) Broaden (25.67%) Add (3.21%)
	Push Down Attribute (Total Count: 74)	Simple (62.16%) Complex (28.57%) Formatting (6.76%)	Change (63.51%) Preserve (36.49%)	Narrow (70.21%) Broaden (23.4%) Add (2.13%)
Method	Rename Method (Total Count: 2,158)	Simple (66.22%) Complex (23.17%) Formatting (9.87%)	Change (81.19%) Preserve (18.81%)	Narrow (36.42%) Broaden (31.16%) Add (24.14%)
	Extract Method (Total Count: 1,694)	Simple (52.42%) Complex (43.15%) Formatting (3.96%)	Change (85.42%) Preserve (14.58%)	Narrow (64.06%) Broaden (26.12%) Add (4.49%)
	Rename Variable (Total Count: 1,387)	Simple (51.62%) Complex (43.98%) Formatting (3.89%)	Change (87.41%) Preserve (12.59%)	Narrow (49.28%) Broaden (32.86%) Remove (9.17%)
Variable	Rename Variable (Total Count: 3,067)	Simple (87.41%) Complex (12.36%) Formatting (0.23%)	Change (98.89%) Preserve (1.11%)	Add(77.35%) Narrow (14.93%) Broaden (6.17%)
	Extract Variable (Total Count: 305)	Simple (61.64%) Complex (36.72%) Formatting (1.31%)	Change (92.13%) Preserve (7.87%)	Narrow (71.17%) Broaden (19.57%) Add(6.05%)
	Inline Variable (Total Count: 6)	Simple (83.33%) Complex (16.67%)	Change (100%)	Add (66.67%) Narrow(33.33%)

commit. We then analyze this data to understand whether the refactoring that happened before the rename had any effect on the semantic change applied during the rename. Additionally, we perform an analysis of commit messages using LDA and bi/trigrams in an effort to further contextualize the semantic change; using information about why a given refactoring was applied before the rename to help us understand the semantic changes observed during renames applied afterward.

The first observation we make is that renames applied after another refactoring most frequently changed the target name's meaning somehow; the meaning was less frequently preserved. Therefore, we will first look at renames that changed the meaning of the identifier they were applied to. Please refer to 3.1 for a refresher on the semantic change categories. Table 6 highlights the distribution of these change types for elements that undergo a rename after another type of refactoring operation.

We observe that the majority of the name changes were related to a narrowing in the meaning of the name. Generally, a narrowing in the meaning of an identifier name is related to a specialization of functionality. For example, in commit [80], a developer created the method `readImage(width int, height int)` by performing an *Extract Method* operation in order to add "missing functionality". In a subsequent refactoring operation on this method, the developer renames the method to `readZlibImage(width int, height int)` with the message "Added read support for GM8 gmk files" [81]. As can be seen by the message, the developer specializes the method and hence reflects this behavior in the new method name by narrowing its meaning.

The next most common type of semantic change was the broadening of the identifier's name. Developers perform a broadening of the name when they generalize the behavior of the identifier. As an example, in commit [82], a developer performs a *Pull Up Attribute* on `idColumn` as part of generalizing change – "Create generic table class" . Thereafter, the developer renames the attribute to `id` in order to make it consistent with the earlier generalizing task – "Rename generic table column fields" [83]. Finally, adding to the identifier

name was the third most frequent type of semantic change.

There are a few interesting things to point out in Table 6. The first is that a *Rename Variable* followed by another *Rename Variable* tended to add meaning instead of narrow or broaden. The same applies to renames occurring after a *Move Attribute* refactoring and after an *Inline Variable* refactoring. However, these are the only examples of a break from the typical pattern of Narrow being the most common semantic change type. If we only contextualize using refactorings applied before renames, there are few significant differences in the types of semantic changes applied after different types of refactorings. While this data does indicate the popularity of narrowing, adding to, or broadening the meaning of a name, it does not completely help us understand what the developers were trying to accomplish; an *Extract Method* refactoring occurring before a rename does not serve as a strong indicator of what semantic change will happen if a rename is applied afterward.

To help us further contextualize these refactorings and the renames occurring afterward, we perform LDA and n-gram analysis on commit messages associated with the rename refactorings occurring after a refactoring operation. Our previous work also used LDA in a similar context [23], but it performed LDA analysis on the commit message associated with the rename without taking into account if the rename occurred in isolation or immediately after another refactoring. We extend the topic modeling approach in [23] by incorporating additional text preprocessing and the use of topic coherence scores in order to improve the quality of our text analysis compared to the original paper. The results of this analysis are in Tables 7, 8, 9, and 10. In each table, we show the two strongest topics from LDA along with either a bigram or trigram analysis. We present either the bigram or trigram that is the most relevant. Using the data in these tables, we can see some indication of what development activity caused different types of semantic changes when applying a rename.

Table 7 shows data for all method renames that are preceded by a variable rename, and resulted in the name of the method broadening in meaning. These preceded a rename which resulted in a *broaden meaning*. The data here indicates

Table 7: Broadening of a method name after a variable rename

Analysis	**Output**
LDA Topic 1	change (0.090), model (0.086), past (0.068), discussed (0.068), allow (0.019), lambda(0.019), route(0.019), work(0.015), early(0.014), simplified(0.014)
LDA Topic 2	change (0.088), model (0.061), past (0.049), discussed (0.049), fix (0.028), factory (0.026), changed (0.023), loader (0.023), add (0.017), set (0.013)
Trigram	(discussed, past, model), (change, discussed, past), (model, change, discussed), (past, model, change), (discussed, past, added), (changed, loader, factory), (loader, factory, changed), (factory, changed, loader), (location, model, change), (render, nicely, html)

Table 8: Narrowing of a variable name after its extraction

Analysis	**Output**
LDA Topic 1	code (0.091), binding (0.083), data (0.081), updated (0.074), fix (0.028), add (0.025), support (0.017), cr (0.009), custom (0.009), request (0.009)
LDA Topic 2	code (0.067), updated (0.060), binding (0.059), data (0.058), record (0.013), id (0.010), custom (0.010), introduced (0.010), remove (0.010), cr (0.007)
Bigram	(data, binding), (binding, code), (updated, data), (code, updated), (revamped, hibernate),(added, method), (array, fix), (attribute, handle), (binding, warning)

Table 9: Narrowing of an attribute name after its pulled-up

Analysis	**Output**
LDA Topic 1	work (0.094), introduce (0.043), security (0.034), option (0.034), addition (0.034), start (0.018), add (0.018), took (0.018), thread (0.018), ongoing (0.018)
LDA Topic 2	symbol (0.077), table (0.077), work (0.061), unit (0.031), option (0.031), property (0.024), fixed (0.022), hierarchy (0.016), added (0.016), implementation (0.016)
Trigram	(hierarchy, option, reduce), (implemented, hierarchy, option), (option, reduce, code), (reduce, code, duplication), (code, duplication, implemented), (duplication, implemented, hierarchy), (gross, value, gross), (addition, security, addition), (code, added, support), (entity, id, field)

Table 10: Adding meaning to a class name after moving it

Analysis	**Output**
LDA Topic 1	method (0.189), added (0.083), adding (0.072), increased (0.071), incremental (0.071), stub (0.071), anonymous (0.071), truly (0.071), fix (0.013), subset (0.013)
LDA Topic 2	test (0.198), validation (0.043), removing (0.030), enable (0.029), mapping (0.029), upgrade (0.029), failing (0.029), concept (0.029), collection (0.015), contains (0.015)
Trigram	(added, method, adding), (adding, truly, anonymous), (incremental, stub, method), (method, added, method), (method, adding, truly), (stub, method, added), (truly, anonymous, increased), (anonymous, increased, incremental), (cleaned, scorer, removing), (field, tree, context)

changes to a model and changes to a factory. An analysis of the commit messages associated with these topics shows that the updates are due to bug fixes or code optimizations. For example, in commit [84], the broadening of the name is associated with the message "...Made the factory generic", which a broaden meaning rename would logically follow. Table 8 has similar data but for a set of *Extract Variable* refactorings which preceded a *narrowing* of the identifier name meaning via rename. The topics and bigrams here indicate code related to data binding, code updates, and code fixes. Again, we took a look at the commit messages associated with this data and found that most of the data bindings were specific to a certain project in our corpus. In this instance [85], the developer uses a generic message, "Updated data binding code...". Ignoring this set of commits, a majority of the remaining messages were associated with bug fixes.

Table 9 shows the implementation of options and reduction in code duplication which preceded a *narrowing* in meaning. An analysis of the commit messages associated with this table shows that the removal of duplicate [86] and legacy [87] code is a task associated with code cleanup activities. These activities can also range from simple identifier renames [88] to more intensive structural changes [89]. Finally, Table 10 indicates the addition of new methods associated with moving a class to a different location, which preceded an *add meaning* change. Examining these commit messages reveals that methods are added in response to enhancing the existing design of the system after the class is moved and hence contribute to the renaming of the class, such as in the case of [90], where the developer performs a "...Method grouping" in the newly moved class.

Preserve meaning was the least occurring semantic type, and not surprisingly, the frequently occurring terms in these commit messages were not change related. These terms include 'fix', 'test' and 'work'. Generally, such terms are associated with behavior correction. Hence, developers feel that the update they make to the code does not necessarily deviate from the originally expected behavior of the identifier. For example, in [91] as part of updates to the user

interface, the developer performs a *Pull Up Method* operation on the method `calcTotal`. The next update [92] to this method is to address an issue, and as part of this task, the developer renames the method to `calculateTotal` to better represent its intended behavior. A cursory glance at the method shows no changes to the functional behavior exhibited by this method.

Summary for RQ3: Developers frequently change the semantic meaning of an identifier name when performing a rename after a refactoring, rather than preserving it. Most frequently, a rename will change this meaning by narrowing (i.e., specializing) the identifier name it is applied to. While the rationale for some semantic changes can be derived from the commit log in addition to the actions that occurred just prior to the rename, classical ways of analyzing large numbers of commit messages provide only a high-level understanding of this rationale and require significant manual analysis to help us fully understand the rationale. The answer to this RQ is that refactorings, occurring before and after a rename, and commit messages can give us some high-level insight into how names semantically change and why. Still, our data shows that further research using additional software artifacts, and new methods of natural language text analysis for software engineering, are required to provide us with stronger insights.

5.5. RQ4: What structural changes occur when an identifier and its corresponding type are changed together?

From our analysis of 310,309 identifier rename instances, we observe that 17.39% (or 53,962) of identifier renames involve a change to their corresponding type. We are interested in understanding how changes to the type name correlate with modifications to the structure of an identifier name. A breakdown of renames which included a change in data type is shown in Table 3.

First, we look at rename forms (i.e., Simple, Complex, Reordering, and Formatting). As mentioned in Section 3.1, a Simple rename involves a change of a single term between the old and new name of the identifier (e.g., `char[] password` → `byte[] encodedPassword`). The *Rename Variable* refactoring op-

Table 11: Distribution of identifier form types when a change in data type occurs

Form Type Change	**Count (Total: 53, 962)**	**Percentage**
Simple	32,448	60.13%
Complex	21,169	39.23%
Formatting	314	0.58%
Reordering	31	0.06%

eration `String moveCoords` → `Point point`, on the other hand, falls under a Complex rename as more than one term between the old and new identifier name has changed. Reordering involves a change of position of terms (e.g., `RecordId recordId` → `IdRecord idRecord`), while Formatting is due to change of case or the addition/removal of a special character (e.g., `AbstractDropDown dropdown` → `DropDown dropDown`). Looking at the types of rename forms, as shown in Table 11, we observe that approximately 60.13% of data type changes are associated with Simple changes to the identifier's name, while Complex changes account for approximately 39.23%. Formatting and Reordering changes each account for less than 1%.

Additionally, we investigate the extent to which an identifier's name contains the name of its data type to see if the type is generally added or removed as identifiers are changed. Prior work considers the inclusion of a type name in the associated identifier's name as an impediment to software maintenance and code comprehension activities [2]. As some insight into this, it could be argued that, in strongly typed languages, including the name of a type in an identifier's name is redundant due to the type being explicitly present already, and modern IDEs will generally inform the developer of an identifier's type using annotations. Another drawback of this naming approach is that the developer will be forced to rename the identifier when changing its data type (or risk the name becoming out of date). This might be a substantial number of instances throughout the

codebase; not just the statement declaring the identifier. For example, when naming the variable `String tupleString`, the developer appends the name of the data type, `String`, to the identifier's name. For this specific example, we observe that the developer, in a subsequent commit, renames the identifier to `List<String> tuple`. As such, it can be seen that the developer had to adjust the name of the identifier due to the old data type being present in the name.

For each instance of a rename refactoring in our dataset, we check if the old and new name contains its respected data type as part of its name (i.e., the identifier name either starts with, ends with, contains or is an exact case-insensitive match of the name of the data type). First, looking at all rename instances (i.e., renames with and without a data type change), we observe that approximately 83.69% (out of 310,309) of the rename refactorings did not contain the name of the data type as part of the old or new identifier's name. Within this 83.69% of renames, approximately 10% of these renames had a change in data type, while the remaining 90% retain the same data type.

Focusing on the remaining 16.31% (or 50,621) of renames on identifier names which contain the name of their corresponding data-type, we have two groups which are summarized in Table 12: G1) identifiers whose corresponding data type changed (top half of the table with 26,227 rename instances); and G2) identifiers whose corresponding data type did not change (bottom half of the table with 24,394 rename instances). Identifier names in G1 tended to exactly match the name of their type even after being renamed (e.g., `LocationStrategy locationStrategy` → `ElementLocator elementLocator`) 34.86% of the time and, when the name of the type was not originally present, they tended to be changed to exactly match their type during a rename (e.g., `BitRateType bitRate` → `BitRateType bitRateType`) 18.73% of the time. This indicates that when a data type and identifier name are changed in-tandem, there is a tendency to include (or keep) the name of the type within the identifier name.

Identifier names in G2 are similar in that the majority of most frequent cases involve adding (or keeping) the data type name to (in) the identifier name. The primary difference between G1 and G2 is that G1 identifiers tend to be

exact matches; the identifier name and type name are exactly the same. In G2, the type names are most frequently appended to the end of the identifier name; the type name is a substring of the identifier name. An explanation for this difference may be that, since types in G1 were modified in-tandem with identifier names, the identifier names are more intricately linked to the type name. Either by already having included it (in the 34.86% case) or for some other reason (in the 18.73% case). Next, we look at the data in this set of 18.73% to try and determine what these other reasons might be. While there was no visibly generalizable trend, we notice that rename instances in this set contain a mix of primitive and non-primitive data types associated with the original name of the identifier and, as part of the rename process, all primitive data types were converted to non-primitive data types (e.g., `long timestamp` → `Clock clock`). This might indicate that one of the trends for the primitives in this data is that these are a case of broaden-meaning changes, where identifiers with primitive types are made into objects with more data and behavior. Our project webpage [41] contains the rest of the combinations not present in Table 12.

Summary for RQ4: Looking at the 53,962 instances of renames applied to both an identifier and its given type, 60% of these changes are Simple, while 39% are Complex. This contrasts with the general population of renames in our study (i.e., regardless of whether there was a change to the type), where 68% are Simple and 29% Complex (Table 2). Of the 16.31% of identifiers involved in this RQ, most added or preserved their type name during a rename refactoring. A minority removed their type name. We observe that renames which involve a change to the type name tended to also involve identifiers with names exactly matching their type. Whereas, when there was not a data type change with the rename, the type name was a substring and tended to be appended to the end of the corresponding identifier name. Generally, developers tended to add or keep type names during renames rather than remove them. More research is required to ascertain the degree to which type names negatively impact the identifier names that they are a part of, but it is possible to recommend developers reconsider whether there is a reason the type name *should* be part of the

Table 12: Distribution of occurrence for the different scenarios where the name of the data type is present in the identifier's name

Old Identifier Name	New Identifier Name	Count	Percentage
Renames with data type changes that contain the name of the data type in the identifier's name (Total Count: 26,227)			
Exact match	Exact match	9,143	34.86%
Does not contain	Exact match	4,913	18.73%
Exact match	Does not contain	3,746	14.28%
...other combinations		8,425	32.12%
Renames without data type changes that contain the name of the data type in the identifier's name (Total Count: 24,394)			
Does not contain	Exact match	5,470	22.42%
Ends with type name	Ends with type name	4,625	18.96%
Does not contain	Ends with type name	3,447	14.13%
...other combinations		10,852	44.49%

identifier during a rename. The trends in Table 12 are reported more fully in our openly available dataset.

5.6. RQ5: What semantic changes occur when an identifier and its corresponding type are changed together?

To answer this research question, we focus our analysis only on rename refactorings that included a change in data type (i.e., 53,962 or $\approx$ 17.39% of rename instances) and analyze how modifications applied to these names are reflected in their data type.

We examine how the semantic meaning of an identifier varies when there is a change to the associated data type. The majority of semantic updates involved a change in the meaning of the identifier. A drill-down into the change in meaning types shows that developers change the data type when Narrowing the meaning of the name approximately 67.91% of the time (e.g., `Parse parse` $\rightarrow$ `ParseResult parseResult`). A Broadening of the identifier names occurs

Table 13: Examples highlighting covariant and contravariant rename instances

Semantic Change Type	**Rename Instances**
	Covariant
Identifier Name: Narrow	`Mongo mongo` → `MongoClient mongoClient`
Data Type Name: Narrow	`Client client` → `ClientEditor clientEditor`
Identifier Name: Broaden	`TabComponent childTabComponent` → `Tab childTab`
Data Type Name: Broaden	`DateTime availabilityEnd` → `Duration availability`
Identifier Name: Preserve	`CsvCreator csvCreator` → `CsvMaker csvMaker`
Data Type Name: Preserve	`Log log` → `Logger logger`
	Contravariant
Identifier Name: Narrow	`SolrConfig solrConfig` → `String solrConfigFile`
Data Type Name: Broaden	`GraphRoute graphRoute` → `Object graphRouteObj`
Identifier Name: Broaden	`String fileName` → `File file`
Data Type Name: Narrow	`Executor workerPool` → `ExecutorService pool`
Identifier Name: Preserve	`String validationInformation` → `Message validationInfo`
Data Type Name: Narrow	`QueryOption reusable` → `QueryOptionReuse reuse`

20.98% of the time (e.g., `String jobName` → `Job job`), followed by Preserve at 8.80% (e.g., `FormulaContext formula` → `ExpressionContext expression`), Add and Remove at 1.62%, and 0.64%, respectively. This contrasts somewhat with our findings on general renames (RQ3), because in RQ5 we find that these renames tend to narrow meaning more often (+23% more often), add meaning less often (-36% less often), and broaden meaning more often (+5%) compared to general rename semantic changes examined in RQ3. If we look at semantic updates made directly to the type name, approximately 69% (or 27,298) of the data type changes show a narrow in meaning, while 24% broadened with the remaining 7% belonging to add and remove.

We also look into how the semantic meaning of types and their corresponding identifier names covary. 71.94% (or 28,341) of the identifier and data type name changes show a covariant relationship; both the identifier and its associated data type name underwent the same semantic update. From a more granular view, we observe that the narrowing of an identifier and data type name occur the most, approximately 56.28% (or 22,171). An example of this type of occurrence is when the developer performs the following *Rename Attribute* operation: `DateFormat defaultDateFormat` → `DateTimeFormatter defaultDateTimeFormatter`. In this example, both the identifier name and data type undergo a narrowing of its respective original meaning. The next two highest occurrences were contravariant: a narrowing of the identifier name and broadening of the data type name at 12.64%; and covariant: a broadening of both the names at 11.02%. In Table 13, we provide examples of covariant and contravariant instances that occurred in our dataset.

Finally, we look at the relationship between identifier names being changed to/from plural form and their data type changing to/from a collection. To detect a change in plurality, we compare the matched terms in the old and new identifier names looking for either a singular to plural or plural to singular change between the matched terms. For example, when the developer renames the attribute `defaultValue` to `defaultValues`, the part of speech for the term 'Value' changes from singular to plural. At a high level, as shown in Table 14,

Table 14: Mapping between identifier name change in plurality and change in data type

Change in Data Type?	**Change in Plurality?**	**Count (Total: 310,309)**	**Percentage**
No	No	252,940	81.51%
Yes	No	50,840	16.38%
No	Yes	3,407	1.10%
Yes	Yes	3,122	1.01%

the majority of renames did not undergo a data type change nor a change in the plurality of their name. However, if we were to focus on only instances that show a change in plurality, approximately 47.82% (3407/(3407 + 3122)) of plurality changes also had a change in data type (e.g., `List<String> contextNames` → `String contextName`), while the other 52.18% (3122/(3407+3122)) of plurality renames did not have a change of data type (e.g., `String hostName` → `String hostNames`).

We also detect when data types that were part of a rename were changed to or from a collection type. Table 15 provides a breakdown of the various combinations of single-reference and collection-based data types that underwent a change in data type. Our analysis shows that the majority of type changes (Table 15, ≈ 82.64%) were not group/collection based (i.e., neither the old or new name utilized an array or collection-based data type). Identifiers that did utilize collection based data types in either the new, old, or both names (e.g., `List<String> contextNames` → `String contextName`) accounted for around 17.36%.

We use the data about plurality and data-type above to study how identifier name plurality and data-type are connected. We observe that around 69.47% of renames that did not have a change in plurality (but did have a type change; Table 16) also did not utilize collection-based data types in either

the old or new name (e.g., `DateTime date` → `LocalDate day`). Additionally, around 3.74% (Table 16, $(900 + 1120)/53962$) of the instances whose data type changed to a collection-based data type change did not show a change in plurality. For example, even though the *Rename Attribute* refactoring: `String exportToolCommand` → `List<String> executableCommand` does not show an overall change in plurality, the developer performs a change in data type by moving from a non-collection to a collection based data type. When a data-type is modified such that it becomes a collection, 64.29% (Table 16, $1621/(900+1621)$) of the time there is a change in plurality for its corresponding identifier name and 35.7% of the time, there is no change in the plurality of the name. When a data-type is modified such that it ceases to be a collection, 53.02% (Table 16, $1264/(1264 + 1120)$) of the time there is a change in plurality for the corresponding name and 46.98% of the time, there is no change in plurality. One other interesting note about this table is that 13.17% of the time, when there was a change in type during a rename operation, the plurality of the identifier changed but we did not detect a collection type. This indicates that the objects' class may have internally changed to include some form of collection or collection-like behavior, which we would not be able to detect since we only look at type signatures without doing internal analysis on classes.

Summary for RQ5: From a semantic perspective, consistent with RQ3, we observe that developers generally narrow the name of the identifier in conjunction with a change in data type as opposed to other types of semantic change types. However, the data also shows that this frequency is more pronounced (i.e., higher) for renames which involve type changes. We also note that there was a decrease in *add meaning* changes and a slight increase in *broaden meaning* changes compared to the general set of renames from RQ3. Additionally, when a data-type is modified such that it becomes a collection, 64.29% of the time there is a change in plurality for its corresponding identifier name, and 35.7% of the time, there is no change in the plurality of the name. When a data-type is modified such that it ceases to be a collection, 53.02% of the time, there is a change in plurality for the corresponding name, and 46.98% of the time, there is

Table 15: Distribution of data type changes with primitive/non-primitive and single/collection data types for rename instances that changed data type

Old Data Type	New Data Type	Count	Percentage
Primitive vs. Non-Primitive (Total Count: 53,862)			
Non-Primitive	Non-Primitive	49,380	91.51%
Primitive	Non-Primitive	2,532	4.69%
Non-Primitive	Primitive	1,157	2.14%
Primitive	Primitive	893	1.65%
Single vs. Collection (Total Count: 53,962)			
Single	Single	44,593	82.64%
Collection	Collection	4,464	8.27%
Single	Collection	2,521	4.67%
Collection	Single	2,384	4.42%

Table 16: Mapping between identifier name change in plurality and use of collection-based data type for rename instances that underwent a change in data type

Is Data Type a Collection?		Change in	Count	%
Old Identifier	New Identifier	Plurality?	(Total: 53,962)	
No	No	No	37,487	69.47%
No	No	Yes	7,106	13.17%
No	Yes	No	900	1.67%
No	Yes	Yes	1,621	3.00%
Yes	No	No	1,120	2.08%
Yes	No	Yes	1,264	2.34%
Yes	Yes	No	3,556	6.59%
Yes	Yes	Yes	908	1.68%

no change in plurality. 1.68% of the time, the data type is already a collection object, and the identifier is modified to be plural to reflect this. Finally, we found that most identifier names covariantly evolve with their corresponding type name, and a minority of the renames we examined showed a contravariant relationship.

5.7. RQ6: What refactorings most frequently appear before and after an identifier and its corresponding type are changed together? Are there specific semantic changes which correlate with these refactorings?

To answer this question, we look at the refactorings that surround attribute, method, parameter, and variable rename refactorings that have a change in data type. Hence, the input data for this research question is a subset of the dataset used in RQ2; specifically the subset of renames which included a change in type. Except for class, we extract the subset of rename instances for attributes methods, and method variables that underwent a change in data type while being renamed. In total, 283 ($\approx$ 15.31%) attribute renames that underwent a data type change also had a refactoring occurring either before or after the rename. Similarly, we observe 564 ($\approx$ 9.63%) variable, and 734 ($\approx$ 6.78%) method renames under the same criteria.

Similar to RQ2, the majority of the refactorings occurred before a rename refactoring. Hence, we look at the refactorings that frequently occurred before a rename with a data type change. For variables, we observe that the majority of variable renames containing a data type change occurred approximately 42.73% of the time after the same variable was previously renamed. Rename-based data type changes for methods occurred 20.30% of the time after an *Extract Method* operation, and 16.89% of the time after a *Rename Variable* within the same method. This is nearly identical to RQ2 data (Table 5), where *Extract Method* and *Rename Variable* occurred 18.53% and 15.17% of the time, respectively, before a rename. Likewise, renames occurred after *Move Attribute* $\approx$ 66.08% of the time. This relationship is weaker than in Table 5, where renames occurred after *Move Attribute* 83.32% of the time.

Finally, we investigate the semantic updates made to the identifier's name, which follows a refactoring operation. Presented in Table 17, are the top three refactoring operations that preceded a rename refactoring that also had a data type change. This table also shows the distribution of semantic updates that the name undergoes. The trends mirror what we discussed in RQ4 and RQ5, but are broken down by refactoring which preceded the rename of an identifier name and its type. Add-meaning changes were much less likely when an identifier and its type are renamed together. If we compare Table 6 and Table 17, we can see that general identifier renames with a preceding *Move Attribute* refactoring tend to add meaning, but when we narrow to identifier renames which change the type in-tandem, we see a sharp decline in the relative number of add-meaning changes (a reduction from 54% to 1.07%) and instead see a majority of narrow and broadening-meaning changes. A similar drop occurs for identifier renames with a preceding *Rename Variable* (from 77% to <3%). This data breaks down some of the trends we note in RQ5; showing us that, for example, the loss of add-meaning changes has some context (i.e., *Move Attribute* when an identifier and its type are renamed) which may be leveraged when understanding, or recommending/suggesting, renames. The dataset for this study, available on our project website [41], contains the entire list of before and after refactorings.

Summary for RQ6: Comparing the refactoring co-occurrence data from RQ2 with RQ6, our findings from RQ6 are similar to our RQ2 findings in that the refactorings occurring before the rename are more or less the same (i.e., *Rename Variable*, *Move Attribute*, and *Extract Method*). However, we also find that the relationships with these refactorings in RQ6 are generally weaker or roughly the same as in RQ2. This indicates that a rename in which a data type is changed may be less likely to have a co-occurring refactoring. In RQ5, we found that narrow- and broaden-meaning changes are emphasized while add-meaning is de-emphasized when an identifier and its type change together versus general renames. In RQ6, we further broke this trend down and see that the reduction, while pervasive, heavily affects refactoring contexts, as we can see if we compare semantic changes made to renames correlated with *Move Attribute*

Table 17: An overview of the types of semantic updates an identifier name with a data type change undergoes when preceded by another refactoring operation

Identifier Type	**Refactoring Before Rename**	**Top 3 Types of Rename Forms**	**Type of Semantic Update**	**Top 3 Semantic Change Subtypes**
Attribute	Move Attribute (Total Count: 187)	Simple (65.57%) Complex (36.90) Formatting (0.54%)	Change (90.37%) Preserve (9.63%)	Narrow (77.01%) Broaden (41.71%) Add (1.07%)
	Pull Up Attribute (Total Count: 61)	Simple (54.1%) Complex (44.26%) Formatting (1.64%)	Change (91.80%) Preserve (8.20%)	Narrow (54.46%) Broaden (26.23%) Remove(6.56%)
	Push Down Attribute (Total Count: 16)	Simple (56.25%) Complex (43.75%)	Change (87.5%) Preserve (12.5%)	Narrow (68.75%) Broaden (18.75%)
Method	Extract Method (Total Count: 149)	Simple (56.38%) Complex (43%) Formatting(0.67 %)	Change (91.28%) Preserve (8.72%)	Narrow (62.42%) Broaden (24.83) Add(0.67%)
	Rename Variable (Total Count: 124)	Complex (50.81%) Simple (47.58%) Formatting(0.81%)	Change (91.13%) Preserve (8.87%)	Narrow (66.94%) Broaden (21.77%)
	Rename Method (Total Count: 105)	Simple (67.62%) Complex (32.38%)	Change (83.81%) Preserve (16.19%)	Narrow (45.71%) Broaden (34.29%) Add(2.86%)
Variable	Rename Variable (Total Count: 241)	Simple (56.85%) Complex (43.15%)	Change (97.1%) Preserve (2.9%)	Narrow (59.75%) Broaden (32.37%) Remove (3.73%)
	Extract Variable (Total Count: 95)	Simple (64.2%1) Complex (33.68%) Formatting (2.11%)	Change (90.53%) Preserve (9.47%)	Narrow (70.53%) Broaden (19.95%) Remove (1.05%)
	Replace Variable With Attribute (Total Count: 3)	Complex (66.67%) Simple (33.33%)	Change (100%)	Narrow (66.67%) Broaden (33.33%)

in Table 6 with the same in Table 17 and note the significant drop in add-meaning changes (a reduction from 54% to 1.07%). This data indicates that renames which include type changes may need to be treated as special cases in any future rename recommendation/analysis effort due to the relationship between the identifier and its corresponding type.

6. Discussion

In this paper, we extend our prior work on contextualizing renames [24, 23] by exploring renames which involve a change in data type. We focus on this set of renames for three reasons: 1) changes to an identifier's type are relatively easy to detect in many programming languages. Therefore, making suggestions to developers on the fly when a type change is performed is already feasible in modern IDEs. 2) Types have strong influence over the data and behavior represented by an identifier, so changes to the type can have heavy significance on their associated identifier. 3) Type changes are a simple way for us to explore some non-refactoring code changes related to renames. Type changes provide another dimension with which to understand how names evolve and why. These results will provide insight for our long term goals. Further, there are a sizeable number of renames with data type changes; we observed 53,962 instances (or $\approx 17.39\%$) of the total set (minus class renames since class names have no type). Analyzing these changes is more straightforward than studying renames that did not undergo a data type change.

The data from our RQs reports some very interesting trends in the practice of renaming identifiers as well as their corresponding data types. These results have particular significance in the context of recommending when and how to rename. Below, we discuss how our results can directly or indirectly impact our understanding of renames and the eventual recommendation of when/how to rename. We also discuss the significant challenges and work required to augment, make our findings actionable, and integrate them into a development environment. These problems fall into two categories: analyzing surrounding

code changes and analyzing commit messages and other natural language documents. We discuss these in Section 6.2 and 6.3.

6.1. Takeaways and Actionable Results

6.1.1. Takeaways from RQ1

RQ1 shows us that developers with a relatively smaller amount of experience than their peers have a higher likelihood of applying rename refactorings than other types of refactorings. While it is known that developers refactor code to remove smells [16], work by Palomba et al. [93] shows that the developer's experience and knowledge of the system plays a crucial role in the resolving of smells. Further, work by Kim et al. [94] shows that developers perform more non-rename refactorings manually than rename refactorings. Hence, our findings that experienced developers are more favorable to performing non-rename operations align with these prior studies. Furthermore, using tools to perform automated renames, while productive, runs the risk of introducing linguistic anti-patterns [95, 17] in the code. *Our findings from RQ1 give us data on our target audience. Renames are applied more frequently by developers with less experience, potentially due to how simple it is to apply them. When a rename is applied, suggestions about other names related to the current name should be highlighted, as should linguistic anti-patterns and the reasoning behind the suggestion. Developers with less experience may not be familiar with as much of the system and, thus, may benefit more from increased, minimally-intrusive, direction.*

6.1.2. Takeaways from RQ2, RQ3, and RQ6

Data from RQ2, RQ3, and RQ6, while retrospective in nature (i.e., we analyze post-rename data such as commit messages), help us understand the structure and semantics behind a rename; why a given name changed in the ways that it did. This retrospective data allow us to pinpoint directions for future research by highlighting trends and data points that we do not understand. RQ2s and RQ6s data on refactorings, which precede renames, can be directly used to

recommend renames after certain refactorings are applied. This highlights potential future research directions focusing on the relationship between renames and those specific refactorings (i.e., *Move Class*, *Move Attribute*, and *Extract Method*). For example, in RQ2, we saw that *Move Attribute* often preceded a rename by one commit (71% of the time) and five commits (83.32% of the time); it would make sense to consider recommending a rename after a *Move Attribute*. *The relationships in the data we obtained were never 100%; meaning that it would be inappropriate to always recommend a rename after these refactorings. Given this, we believe that future studies should focus on how to acutely determine when to recommend a rename after these refactoring operations and what type of rename to recommend. Such studies might help highlight why these refactorings are more likely to co-occur with renames than other refactorings and uncover other potential related situations where a rename should be recommended. Finally, data from these RQs also highlight that non-refactoring changes are more common than refactoring changes before, and after, a rename. This means that our future work must deal with code changes that are likely not taxonomized in the same way refactorings are.*

RQ3s data on commit messages highlights that we can obtain high-level development motivations (e.g., add functionality, bug fix) for renaming an identifier, but also that there are significant challenges in obtaining project-specific development actions via commit messages (e.g., adding request handler). *Finding ways to gather this more specific data can help recommendation systems by giving researchers more insight into how developers mentally model changes to the structure and semantics of a name during the rename process. For example, we can understand more about how a developer decides what words to change in an identifier when adding a specific type of functionality (i.e., as opposed to just the general idea of adding functionality) or when extracting a method for a project-specific reason (e.g., extracting a parse function from the request handler). These insights can influence the design of recommendation tools by tailoring them to granular, real-world situations.* We discuss more about the challenges that RQ2, RQ3, and RQ6 highlight in Sections 6.2 and 6.3. Addi-

tionally, our data indicate that renames frequently happen with other renames and that the commits for these frequently contain the term *revert*. A deeper dive into what these reverts are and if they are related to the name is required.

6.1.3. Takeaways from RQ4 and RQ5

The results from RQ4 show us that an overwhelming majority of renames that involve data types are not formatting or term-reorderings. Of the 16.31% of identifiers which contained the name of their type either before or after a rename, most added or preserved their type name during a rename refactoring. A minority removed their type name after a rename was applied, meaning that most renames involving identifiers containing a type name either preserve or add the type name as opposed to removing. Additionally, identifier names that end with the name of the data type will most likely contain the name of the data type at the end of its name (rather than in the middle or at the beginning) when the identifier is renamed. *The inclusion of a type name in an identifier is generally considered poor naming practice [2]. There is an opportunity here to both further study the (dis)advantage of including type names in identifier names, particularly in strongly-typed programming languages, and discourage (or question) the inclusion of a type name when a developer performs a rename. Our data indicate that it is less common to remove type names from identifier names than to preserve or add them, meaning that it is not common practice to avoid including types in identifier names. This recommendation can be readily made in modern IDEs after being verified by researchers to determine when, if ever, a type name should appear in an identifier name. If there are situations where the type name should appear within its corresponding identifier's name, our data highlights where developers commonly include the type within the identifier name.*

The results in RQ5 contrast somewhat with RQ3 because in RQ5 we find that renames on identifiers which were modified with their types tend to narrow meaning more often (+23% more often), add meaning less often (-36% less often), and broaden meaning more often (+5%) compared to general rename se-

mantic changes examined in RQ3. RQ5 also explores the introduction/removal of plurality in identifier names and collections in their corresponding data types. We find that most of the time, when an identifier's type becomes a collection, its name is changed to plural 64.29% of the time. Additionally, when a data type ceases to be a collection, the plurality of its corresponding identifier changes 53.02% of the time. This means that there is a potential linguistic anti-pattern (as defined in [17]) being introduced. Specifically, the plurality of the identifier name may no longer be an accurate reflection of the type. As an example, in [37] the developer renames the public method *getCompilationsUnit* → *getCompilationUnit* (a change in plurality rename). However, in another class that calls this method, the developer does not update the name of the variable *compilationsUnit* that holds the results of the method call. This is an easy mistake to make. Leveraging plurality in the name of identifiers to clarify the use (or non-usage) of collection types is a simple but effective method for conveying semantic information, and current rename tools do not offer these types of suggestions. Our data show that there is a strong opportunity here to introduce useful suggestions to developers when plurality or collection changes are detected. This would help developers avoid making simple mistakes and oversights which could significantly degrade comprehension over time.

Finally, RQ5 finds that co-variance is the most common way for type names an identifier names to evolve together (71.94% of cases), but that there are situations were type names and identifier names are contravariant (28.06% of cases). *Our study highlights the fact that more than half the time, the plurality of a name is updated in response to its type changing to/from a collection. Similar to recommending against including type names in identifier names, recommending a plurality change when a type is being updated to/from a collection is a simple way to make this practice consistent or to ask developers to consider whether it is appropriate. This is another recommendation that can be supported by IDEs and further verified by researchers. In addition, the data on contravariance and covariance supports RQ4 by specifying how type names should evolve within an identifier's name in potential cases where future research determines they should*

appear.

6.2. Challenge 1: Analyzing Renames and General Code Changes

While refactorings [16] are some of the most well-known, taxonomized source code changes, many other types of source code changes are not taxonomized beyond low-level software differencing change-types (e.g., insert, remove, delete operations). *With refactorings, we can examine a cohesive, understood change (i.e., the refactoring itself) and analyze how that change relates to a rename. When a change is not a refactoring but involves multiple statements, it is more difficult to: 1) understand whether those multiple changes are actually related; they might be incidental. And, 2) understand the reasoning/motivation behind those changes; it cannot be easily determined if they represent a bug fix or the addition of new functionality, for example.*

The results from RQ2 and RQ6 indicate that the vast majority of renames have no refactoring before or after their application. This means that to understand more about how changes surrounding these renames affect the renames themselves; we must at least partially solve the two problems above such that we are able to understand the code changes which are related to a given rename. There are some potential paths toward remediating this problem. Change taxonomy studies by Fluri et al. [96, 97] discuss a taxonomy which they derive using ChangeDistiller [98]; a technique for extracting changes using differences. There are other taxonomies for code changes, typically more specific than Fluri's taxonomy. These taxonomize changes which indicate bugs [99, 100], others mine code changes to detect emergent change patterns [101], or present methods for clustering similar code changes or studying repetitiveness [102, 103]. These papers represent a strong start in the direction of a more acute understanding of non-refactoring code changes.

6.3. Challenge 2: Analyzing Rename Commit Messages

Analyzing commit messages also posed a challenge. In particular, we faced many issues in deriving rename motivations. Our automated analysis was able

to determine high-level motivations such as 'modified functionality' since this occurs in all projects. However, to understand changes made to the name as part of 'modified functionality', we also need to know project-specific details about what functionality was added and why. For example, if we determine that 'modified functionality' also involved 'combining two functions' into one, we could draw better insights with respect to how the name evolved. We attempted to automatically derive these motivations from commit messages with only some success; other natural language software artifacts, and general source code changes, might be more useful. The most significant problems we faced with analyzing large numbers of commit messages is that: 1) the terms frequent enough to be detected by LDA/Ngrams are high-level and not descriptive of individual project efforts (e.g., we can determine that projects are performing structure changes, but not what types of structural changes or why). Also, 2) the commit messages sometimes do not contain enough information, potentially indicating the need for more natural language software artifacts, some of which will likely be more challenging to analyze automatically. Therefore, whether analyzing commit messages or other natural language texts, *an effective method for performing natural language analysis on software documents that addresses point # 1 above would improve our ability to understand how names semantically evolve (and how developers mentally model this evolution) by allowing us to determine the causes behind certain semantic changes via analyzing natural language text in commit messages. This would also help us address point # 2, which requires the exploration and analysis of other types of natural language text outside of commit messages. The work we present in this paper shows that this context is obtainable, but there are still significant challenges to it.*

To help support the intuition that commit messages can be used to determine rename motivation, we manually looked at some of the data (150 commits). This helps us determine whether the information we need is contained within commit data, and provide some direction for future work. The results of this manual investigation are shown in Table 18. All categories in this table were identified as causes for renames which involved type changes.

Table 18: Development actions which caused identifier renames.

Category	Definition
Combine Behavior	Two classes collapsed into one class or one class deleted and the other class now does the deleted class' job
Split class	Two classes created from one super class
Add Interface	Interface or abstract class was added during rename of type
Broaden Behavior	Behavior of renamed object has a larger domain after the rename operation
Pure Rename	Rename applied for non-functional purposes
Narrow Behavior	Behavior of renamed object has a smaller domain after rename
Add Behavior	Behavior was added via addition of new code
Modify Behavior	Behavior was updated by some combination of narrow, broaden, and add behavior

In renames which did not involve type changes, only **bolded** categories were identified as causing renames during the manual investigation. For instance, in [104], we noticed that the developer renaming the attribute `InvTweaks instance` → `InvTweaksRunnable tickRunnable` does so to utilize threads in the code to cause delays as a means of working around a certain limitation. This is considered part of the *add behavior* category. In [105], the developer renames the method variable `MetadataProvider metadataProvider` → `PersistenceExtensionFeatureResolver persistenceExtensionFeatureResolver` to better reflect its behavior; no other changes to the code, associated with the identifier, were made by the developer. This is a *pure rename*. As one last example, the change in [106] renames the method variable `MembershipCreator membershipCreator` → `Membership membership` due to the behavior of the original data type (and some other types) being incorporated into a new data type. We consider this a *Combine Behavior*. These tables show the intention behind renames based on manually-examined code changes and commit messages. One interesting, if logical, trend is that renames that involve type changes have a larger number of factors which caused the change (i.e., all eight categories in the table, whereas renames which did not involve a type involved only the bolded categories in our investigation). Part of this is due to some categories implying a change in type name (e.g., split class implies the introduction of a new type and/or rename of old type).

While this data is interesting and further supports the fact that analyzing code/commit messages can help us understand the intention behind identifier and type name modifications, further study is required to create a more formal and exhaustive set of the causes of renames. *However, we highlight that, based on the data in Table 18, a combination of commit messages and code change taxonomy could be very useful. This is apparent in the fact that the categories in this table, which were derived from data in commit messages, allude to types of code change (e.g., splitting a class, adding an interface) and semantic changes (e.g., narrowing an object's behavior). Therefore, this analysis appears to support our assertions above.*

Other work has similarly shown how specialized terminology indicates developer refactoring activities [107]; this also supports the idea that a non-trivial number of commit messages contain enough data to perform this analysis and motivates the need for future research in natural language techniques which are more effective at analyzing commit messages, and other software artifacts, to solve the two problems discussed above.

7. Threats to Validity

In terms of representativeness, the dataset for our study consists of open-source Java systems. However, even though the projects are well-engineered Java systems [42], the results may not generalize to systems written in other languages. Additionally, the type and quantity of detected renaming refactorings are limited to RefactoringMiner's capabilities. However, RefactoringMiner is currently the most accurate refactoring detection tool [44] and is extensively utilized in research concerning refactorings.

We follow an approach from a similar study for our experiment on developer experience by utilizing project contributions as a proxy for the developer's experience. However, as with many software metrics, this metric is not perfect, and may not always be a suitable experience measure.

This study also includes an analysis of commit messages. Hence, we use a peer-review approach to mitigate bias in deciding the terms to present after our commit message analysis. As part of this review process, the authors review the entire list of generated terms; the decisions made during this process had to be unanimous. Additionally, the authors referred to the entire commit message to confirm the context around the terms of interest.

In terms of our experiments that use co-occurring refactorings (RQ2, RQ3, RQ6), while there can be multiple refactoring operations that can occur in a single source file, in a single commit, it is not possible to determine the order in which the developer applied refactoring operations. The only way to obtain this data is to interview the developer responsible for the commit. However, given

the large-scale nature of our study, such an approach is not feasible. Hence we look at the refactoring operations present in commits that occur just before and after a rename refactoring commit. Further, our study did not correlate refactorings occurring in the same commit.

We utilize NLTK in our study to help detect the semantic updates occurring on an identifier name when the identifier is renamed. This partial reliance on NLTK introduces a threat that some of the conclusions drawn by the semantic change detection algorithm may be inaccurate. We alleviate this threat by thorough testing of the tool, but it is known that tools trained specifically on software engineering data tend to generalize better than tools trained on general natural language data and applied to source code [8, 108]. Unfortunately, there are no platforms and models similar to NLTK that are specialized for software-based lexica.

We detect when collection-semantics (e.g., a List data type) have been added or removed by examining type names. This is a conservative estimation; we never identify any type as being a collection when it is not. However, we may miss the addition of collection-like behavior or attributes when it is added to a class' internals. That is, there may be situations where an identifier's name becomes plural because the class which its type represents has been changed to internally include a collection as an attribute. We do no internal class analysis when detecting identifier name plurality and determining if the identifier's behavior has been changed to include/remove collection-semantics. Finally, our data type change detection strategy excluded methods where the return type of either or both instances was void. Hence, while we did capture a large quantity of type change instances, the results may not necessarily be generalizable to all methods.

8. Conclusions and Future Work

In this paper, we use refactorings, static analysis, data types, and commit messages to understand characteristics of changes applied to identifier names

and to determine if these changes correlate to different developer activities (e.g., narrowing of a name after applying an *Extract Method* refactoring operation). Our long term goal is to support recommendation of when/how to rename identifiers and to understand more about developer naming mental models. This study brings us a step closer to achieving this goal by identifying opportunities for rename recommendations which can already be supported (e.g., plurality of names, inclusion of type names in identifier names), identifying trends which should be further explored in future research (e.g., renames correlated with certain refactorings), and highlighting challenges which future research should overcome in order to provide stronger recommendation support.

In future work, we plan to perform a qualitative study on commits, code changes, and documentation associated with renames. This will allow us to expand and improve on the taxonomy discussed in Section 6 and gather data to address the problems described in Sections 6.3 and 6.2. Specifically, we will use data from this paper and a future qualitative study to investigate more effective means of analyzing commit messages and other natural language software artifacts, and we will investigate the use of software differencing techniques [109, 110] to allow us to analyze general software changes that occur around a rename. Both of these directions are directly motivated by the experiences and outcomes resulting from this work. Finally, based on our discussion of the findings and takeaways of our study (Section 6), our next steps include the implementation of IDE plugins that provide developers with rename candidates based on changes to the identifier's data types—- for example, recommending that the developer should change the plurality of the identifier's name when the data type is changed to/from a collection type. These plugins will leverage some of the stronger relationships from this paper to qualify these relationships using human subjects. The dataset utilized in this study is available on our project website [41].

9. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1850412.

References

[1] T. A. Corbi, Program understanding: Challenge for the 1990s, IBM Systems Journal 28 (2) (1989) 294–306. doi:10.1147/sj.282.0294.

[2] R. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin series, Prentice Hall, 2009.

[3] A. A. Takang, P. A. Grubb, R. D. Macredie, The effects of comments and identifier names on program comprehensibility: an experimental investigation, J. Prog. Lang. 4 (1996) 143–167.

[4] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, M. Beigl, Descriptive compound identifier names improve source code comprehension, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, ACM, New York, NY, USA, 2018, pp. 31–40. doi:10.1145/3196321.3196332.
URL http://doi.acm.org/10.1145/3196321.3196332

[5] J. Hofmeister, J. Siegmund, D. V. Holt, Shorter identifier names take longer to comprehend, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 217–227. doi:10.1109/SANER.2017.7884623.

[6] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? a study of identifiers, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 3–12. doi:10.1109/ICPC.2006.51.

[7] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Exploring the influence of identifier names on code quality: An empirical study, in: Software Main-

tenance and Reengineering (CSMR), 2010 14th European Conference on, IEEE, 2010, pp. 156–165.

[8] D. Binkley, D. Lawrie, C. Morrell, The need for software specific natural language techniques, Empirical Softw. Engg. 23 (4) (2018) 2398–2425. doi:10.1007/s10664-017-9566-5.
URL https://doi.org/10.1007/s10664-017-9566-5

[9] C. D. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, E. Hill, An empirical study of abbreviations and expansions in software artifacts, in: Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019.

[10] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Suggesting accurate method and class names, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 38–49. doi:10.1145/2786805.2786849.
URL http://doi.acm.org/10.1145/2786805.2786849

[11] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, Y. Le Traon, Learning to spot and refactor inconsistent method names, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2019, ACM, New York, NY, USA, 2019.

[12] E. W. Høst, B. M. Østvold, Debugging method names, in: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 294–317. doi:10.1007/978-3-642-03013-0_14.
URL http://dx.doi.org/10.1007/978-3-642-03013-0_14

[13] S. L. Abebe, P. Tonella, Automated identifier completion and replacement, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 263–272. doi:10.1109/CSMR.2013.35.

[14] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, K. Inoue, Recommending verbs for rename method using association rule mining, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 323–327. doi:10.1109/CSMR-WCRE.2014.6747186.

[15] C. D. Newman, A. Peruma, R. AlSuhaibani, Modeling the relationship between identifier name and behavior, in: Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019.

[16] Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] V. Arnaoudova, M. Di Penta, G. Antoniol, Y. Guéhéneuc, A new family of software anti-patterns: Linguistic anti-patterns, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 187–196. doi:10.1109/CSMR.2013.28.

[18] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, Y.-G. Gueheneuc, Repent: Analyzing the nature of identifier renamings, IEEE Trans. Softw. Eng. 40 (5) (2014) 502–532. doi:10.1109/TSE.2014.2312942.
URL https://doi.org/10.1109/TSE.2014.2312942

[19] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, J. I. Maletic, Lexical categories for source code identifiers, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 228–239. doi:10.1109/SANER.2017.7884624.

[20] R. S. Alsuhaibani, C. D. Newman, M. L. Collard, J. I. Maletic, Heuristic-based part-of-speech tagging of source code identifiers and comments, in: 2015 IEEE 5th Workshop on Mining Unstructured Data (MUD), 2015, pp. 1–6. doi:10.1109/MUD.2015.7327960.

[21] D. Binkley, M. Hearn, D. Lawrie, Improving identifier informativeness using part of speech information, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, 2011, pp. 203–206. doi:10.1145/1985441.1985471.
URL http://doi.acm.org/10.1145/1985441.1985471

[22] S. Gupta, S. Malik, L. Pollock, K. Vijay-Shanker, Part-of-speech tagging of program identifiers for improved text-based software engineering tools, in: 2013 21st International Conference on Program Comprehension (ICPC), 2013, pp. 3–12. doi:10.1109/ICPC.2013.6613828.

[23] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, An empirical investigation of how and why developers rename identifiers, in: International Workshop on Refactoring 2018, 2018. doi:10.1145/3242163.3242169.
URL http://doi.acm.org/10.1145/3242163.3242169

[24] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, Contextualizing rename decisions using refactorings and commit messages, in: Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2019.

[25] H. Liu, Q. Liu, Y. Liu, Z. Wang, Identifying renaming opportunities by expanding conducted rename refactorings, IEEE Transactions on Software Engineering 41 (9) (2015) 887–900.

[26] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, Y. Luo, Nomen est omen: Exploring and exploiting similarities between argument and parameter names, in: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 1063–1073.

[27] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Learning natural coding conventions, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014. doi:10.1145/

2635868.2635883.
URL http://doi.acm.org/10.1145/2635868.2635883

[28] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Suggesting accurate method and class names, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 38–49. doi:10.1145/2786805.2786849.
URL http://doi.acm.org/10.1145/2786805.2786849

[29] B. Liblit, A. Begel, E. Sweetser, Cognitive perspectives on the role of naming in computer programs, in: In Proc. of the 18th Annual Psychology of Programming Workshop, 2006.

[30] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Relating identifier naming flaws and code quality: An empirical study, in: 2009 16th Working Conference on Reverse Engineering, 2009, pp. 31–35. doi:10.1109/WCRE.2009.50.

[31] S. Fakhoury, D. Roy, S. A. Hassan, V. Arnaoudova, Improving source code readability: Theory and practice, in: Proceedings of the 27th International Conference on Program Comprehension, ICPC '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 2–12. doi:10.1109/ICPC.2019.00014.
URL https://doi.org/10.1109/ICPC.2019.00014

[32] db/src/main/java/com/psddev/dari/db/sqldatabase.java, https://github.com/perfectsense/dari/commit/88e6556.

[33] hibernate-ogm-core/src/main/java/org/hibernate/ogm/grid/entitykey.java, https://github.com/hibernate/hibernate-ogm/commit/7dcfaed.

[34] Choreoswebserviceproxy/src/test/java/ime/usp/br/proxy/proxycontrollertest.java, https://github.com/choreos/choreos_middleware/commit/f2da1f8.

[35] api/src/main/java/org/openmrs/personaddress.java, https://github.com/openmrs/openmrs-core/commit/fd5ed0d.

[36] jangaroo/jangaroo-compiler/src/main/java/net/jangaroo/jooc/jangarooparser.java, https://github.com/coremedia/jangaroo-tools/commit/7a494f1.

[37] jangaroo/jangaroo-compiler/src/main/java/net/jangaroo/jooc/jangarooparser.java, https://github.com/coremedia/jangaroo-tools/commit/fc54b3f.

[38] src/main/java/com/atomicleopard/webframework/view/json/jsonviewresult.java, https://github.com/3wks/thundr/commit/53aaf15.

[39] src/main/java/com/atomicleopard/webframework/view/json/jsonview.java, https://github.com/3wks/thundr/commit/9b02920.

[40] src/main/java/org/sql2o/query.java, https://github.com/aaberg/sql2o/commit/2f23b11.

[41] Project website, https://scanl.org/.

[42] N. Munaiah, S. Kroh, C. Cabrey, M. Nagappan, Curating github for engineered software projects, Empirical Software Engineering 22 (6) (2017) 3219–3253. doi:10.1007/s10664-017-9512-6.
URL https://doi.org/10.1007/s10664-017-9512-6

[43] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 483–494. doi:10.1145/3180155.3180206.
URL http://doi.acm.org/10.1145/3180155.3180206

[44] L. Tan, C. Bockisch, A survey of refactoring detection tools, in: Software Engineering, 2019.

[45] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, A. Bacchelli, A large-scale empirical exploration on refactoring activities in open source software projects, Science of Computer Programming 180.

[46] D. Silva, N. Tsantalis, M. T. Valente, Why we refactor? confessions of github contributors, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Association for Computing Machinery, 2016.

[47] M. Hucka, Spiral: splitters for identifiers in source code filesdoi:10.21105/joss.00653.
URL https://doi.org/10.21105/joss.00653

[48] S. Bird, E. Klein, E. Loper, Natural language processing with Python: analyzing text with the natural language toolkit, "O'Reilly Media, Inc.", 2009.

[49] G. A. Miller, Wordnet: a lexical database for english, Communications of the ACM 38 (11) (1995) 39–41.

[50] Primitive data types, https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html, (Accessed on 11/11/2019).

[51] Collections framework overview, https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html, (Accessed on 11/11/2019).

[52] Chapter 14. blocks and statements, https://docs.oracle.com/javase/specs/jls/se13/html/jls-14.html#jls-14.8, (Accessed on 11/11/2019).

[53] src/test/java/stormpot/countingallocator.java, https://github.com/chrisvest/stormpot/commit/459d423.

[54] src/test/java/stormpot/countingallocatorwrapper.java, https://github.com/chrisvest/stormpot/commit/d2931d3.

[55] apvs/src/main/java/ch/cern/atlas/apvs/client/ui/abstractmeasurementview.java, https://github.com/cern/apvs/commit/c1e5792.

[56] apvs/src/main/java/ch/cern/atlas/apvs/client/ui/measurementview.java, https://github.com/cern/apvs/commit/71fc572.

[57] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, Journal of machine Learning research 3 (Jan) (2003) 993–1022.

[58] M. Röder, A. Both, A. Hinneburg, Exploring the space of topic coherence measures, in: Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, ACM, New York, NY, USA, 2015, pp. 399–408. doi:10.1145/2684822.2685324.
URL http://doi.acm.org.ezproxy.rit.edu/10.1145/2684822.2685324

[59] A. Barua, S. W. Thomas, A. E. Hassan, What are developers talking about? an analysis of topics and trends in stack overflow, Empirical Software Engineering 19 (3) (2014) 619–654. doi:10.1007/s10664-012-9231-y.
URL https://doi.org/10.1007/s10664-012-9231-y

[60] D. Jurafsky, J. H. Martin, Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition, Prentic e Hall.

[61] D. E. Krutz, N. Munaiah, A. Peruma, M. Wiem Mkaouer, Who added that permission to my app? an analysis of developer permission changes in open source android apps, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2017, pp. 165–169. doi:10.1109/MOBILESoft.2017.5.

[62] de.prob.units/src/de/prob/units/sc/contextattributeprocessor.java, https://github.com/hhu-stups/prob-rodinplugin/commit/32601b5.

[63] mes-core/mes-core-data/src/main/java/com/qcadoo/mes/core/data/internal/dataaccessserviceimpl.java https://github.com/qcadoo/mes/commit/15a2615.

[64] Z. Xing, E. Stroulia, Refactoring detection based on umldiff change-facts queries, in: 2006 13th Working Conference on Reverse Engineering, 2006, pp. 263–274. doi:10.1109/WCRE.2006.48.

[65] H. Li, S. Thompson, Let's make refactoring tools user-extensible!, in: Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 32–39. doi:10.1145/2328876.2328881.
URL https://doi.org/10.1145/2328876.2328881

[66] M. Mohamed, M. Romdhani, K. Ghédira, Classification of model refactoring approaches, Journal of Object Technology 8 (6) (2009) 121–126.

[67] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18. doi:10.1109/TSE.2011.41.

[68] src/main/java/com/stripe/model/threedsecure.java, https://github.com/stripe/stripe-java/commit/4fdadaf.

[69] src/main/java/com/stripe/model/applepaydomain.java, https://github.com/stripe/stripe-java/commit/19d4d5a.

[70] src/main/java/org/atlasapi/application/applicationconfiguration.java, https://github.com/atlasapi/atlas-model/commit/4da9fc2.

[71] src/main/java/org/atlasapi/application/applicationconfiguration.java, https://github.com/atlasapi/atlas-model/commit/fc19c98.

[72] heroku-api/src/main/java/com/heroku/api/command/login/basicauthlogincommand.java, https://github.com/heroku/heroku.jar/commit/008dbc2.

[73] heroku-api/src/main/java/com/heroku/api/command/login/basicauthlogin.java, https://github.com/heroku/heroku.jar/commit/0c1c18d.

[74] core/src/main/java/org/mapfish/print/processor/map/createmapprocessor.java, https://github.com/mapfish/mapfish-print/commit/bc1f422.

[75] core/src/main/java/org/mapfish/print/http/httprequestcache.java, https://github.com/mapfish/mapfish-print/commit/fe44bd1.

[76] qtiworks/web/controller/instructor/instructorassessmentmanagementcontroller.java, https://github.com/davemckain/qtiworks/commit/2a1f9df.

[77] qtiworks/web/controller/instructor/instructorassessmentmanagementcontroller.java, https://github.com/davemckain/qtiworks/commit/9cb51b2.

[78] src/main/java/se/crafted/chrisb/ecocreature/drops/sources/abstractdropsource.java, https://github.com/mung3r/ecocreature/commit/42e5d9f.

[79] src/main/java/se/crafted/chrisb/ecocreature/drops/sources/abstractdropsource.java, https://github.com/mung3r/ecocreature/commit/3e2f216.

[80] Lateralgm/org/lateralgm/file/gmstreamdecoder.java, https://github.com/ismavatar/lateralgm/commit/2d1bdaf.

[81] org/lateralgm/file/gmstreamdecoder.java, https://github.com/ismavatar/lateralgm/commit/e41c4c5.

[82] jack-store/src/com/rapleaf/jack/store/jstable.java, https://github.com/liveramp/jack/commit/762b540.

[83] jack-core/src/com/rapleaf/jack/queries/generictable.java, https://github.com/liveramp/jack/commit/b331247.

[84] src/test/java/org/motechproject/ananya/kilkari/handlers/obdrequesthandlertest.java, https://github.com/motech/ananya-kilkari/commit/b3b95f4.

[85] name.abuchen.portfolio.ui/src/name/abuchen/portfolio/ui/util/bindinghelper.java, https://github.com/buchen/portfolio/commit/1bdeccb.

[86] vertx-core/src/main/java/io/vertx/core/http/httpclientoptions.java, https://github.com/eclipse-vertx/vert.x/commit/921c69e.

[87] nuget-tests/src/jetbrains/buildserver/nuget/tests/server/trigger/packagecheckertestbase.java, https://github.com/jetbrains/teamcity-nuget-support/commit/da10d2c.

[88] qtiworks-engine/src/main/java/uk/ac/ed/ph/qtiworks/domain/entities/candidateevent.java, https://github.com/davemckain/qtiworks/commit/0c924ab.

[89] org.jrebirth.af/core/src/main/java/org/jrebirth/af/core/component/basic/innercomponentbase.java, https://github.com/jrebirth/jrebirth/commit/d82fb1b.

[90] https://github.com/unquietcode/flapi/commit/4586325.

[91] abuchen/portfolio/ui/dialogs/transactions/buysellmodel.java, https://github.com/buchen/portfolio/commit/9fc2fad.

[92] /abuchen/portfolio/ui/dialogs/transactions/abstractsecuritytransactionmodel.java, https://github.com/buchen/portfolio/commit/e1d7472.

[93] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 101–110. doi:10.1109/ICSME.2014.32.

[94] M. Kim, T. Zimmermann, N. Nagappan, An empirical study of refactoringchallenges and benefits at microsoft, IEEE Transactions on Software Engineering 40 (7) (2014) 633–649. doi:10.1109/TSE.2014.2318734.

[95] V. Arnaoudova, M. Di Penta, G. Antoniol, Y. Guéhéneuc, A new family of software anti-patterns: Linguistic anti-patterns, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 187–196. doi:10.1109/CSMR.2013.28.

[96] B. Fluri, H. C. Gall, Classifying change types for qualifying change couplings, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 35–45. doi:10.1109/ICPC.2006.16.

[97] H. C. Gall, M. Pinzger, B. Fluri, Change analysis with evolizer and changedistiller, IEEE Software 26 (01) (2009) 26–33. doi:10.1109/MS.2009.6.

[98] B. Fluri, M. Wuersch, M. PInzger, H. Gall, Change distilling:tree differencing for fine-grained source code change extraction, IEEE Transactions on Software Engineering 33 (11) (2007) 725–743. doi:10.1109/TSE.2007.70731.

[99] M. Martinez, L. Duchien, M. Monperrus, Automatically extracting instances of code change patterns with ast analysis, in: Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 388–391. doi:10.1109/ICSM.2013.54.
URL https://doi.org/10.1109/ICSM.2013.54

[100] M. Martinez, M. Monperrus, Coming: A tool for mining change pattern instances from git commits, in: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 79–82. doi:10.1109/ICSE-Companion.2019.00043.
URL https://doi.org/10.1109/ICSE-Companion.2019.00043

[101] S. Negara, M. Codoban, D. Dig, R. E. Johnson, Mining fine-grained code changes to detect unknown change patterns, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 803–813. doi:10.1145/2568225.2568317.
URL http://doi.acm.org/10.1145/2568225.2568317

[102] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, M. Philippsen, Automatic clustering of code changes, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp. 61–72. doi:10.1145/2901739.2901749.
URL http://doi.acm.org/10.1145/2901739.2901749

[103] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, H. Rajan, A study of repetitiveness of code changes in software evolution, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 180–190. doi:10.1109/ASE.2013.6693078.
URL https://doi.org/10.1109/ASE.2013.6693078

[104] src/mod_invtweaks.java, https://github.com/mkalam-alami/inventory-tweaks/commit/1dfd242.

[105] java/org/jboss/arquillian/persistence/metadata/metadataprovidertransactionaltest.java, https://github.com/arquillian/arquillian-extension-persistence/commit/1a06974.

[106] jdeeco-core/src/cz/cuni/mff/d3s/deeco/processor/ensembleparser.java, https://github.com/d3scomp/jdeeco/commit/0bc8911.

[107] E. A. Alomar, M. W. Mkaouer, A. Ouni, Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit changes, in: Proceedings of the 3rd International Workshop on Refactoring, ACM, New York, NY, USA, 2019.

[108] R. Jongeling, P. Sarkar, S. Datta, A. Serebrenik, On negative results when using sentiment analysis tools for software engineering research, Empirical Software Engineeringdoi:10.1007/s10664-016-9493-x.

[109] M. J. Decker, M. L. Collard, L. G. Volkert, J. I. Maletic, srcdiff: A syntactic differencing approach to improve the understandability of deltas, Journal of Software: Evolution and Process n/a (n/a) e2226, e2226 JSME-19-0050.R1. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2226, doi:10.1002/smr.2226.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2226

[110] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, M. Monperrus, Fine-grained and accurate source code differencing, in: Proceedings of the

29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 313–324. doi:10.1145/2642937.2642982.
URL http://doi.acm.org/10.1145/2642937.2642982