5-2022

# Why did you clone these identifiers? Using Grounded Theory to understand Identifier Clones

Luis Angel Gutierrez Galaviz
lxg8800@rit.edu

# Why did you clone these identifiers? Using Grounded Theory to understand Identifier Clones

by

**Luis Angel Gutierrez Galaviz**

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Software Engineering

Supervised by
Dr. Christian D. Newman

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
May 2022

The thesis "Why did you clone these identifiers? Using Grounded Theory to understand Identifier Clones" by Luis Angel Gutierrez Galaviz has been examined and approved by the following Examination Committee:

Dr. Christian D. Newman
Assistant Professor
Thesis Committee Chair

Dr. J. Scott Hawker
Associate Professor
Graduate Program Director

Dr. Mohamed Wiem Mkaouer
Assistant Professor

## Acknowledgments

Thanks to my advisor, Dr. Christian D. Newman, for his support and guidance throughout this research project. Dr. Newman was the first to introduce me into this research area dealing with Program Comprehension through studying the relationship between natural language and identifier naming. His vast knowledge in this field was instrumental in the results produced in this research. His passion for software quality and research have taught me valuable lessons that will help me in my career as a Software Engineer.

Thanks to my family, for encouraging me to pursue my Master of Science degree in Software Engineering, and for supporting me from start to finish.

*I would like to dedicate this thesis to my family for their encouragement and support throughout my academic career.*

## Abstract

Developers spend most of their time comprehending source code, with some studies estimating this activity takes between 58% to 70% of a developer's time [1] [2]. To improve the readability of source code, and therefore the productivity of developers, it is important to understand what aspects of static code analysis and syntactic code structure hinder the understandability of code. Identifiers are a main source of code comprehension due to their large volume [3] and their role as implicit documentation of a developer's intent when writing code. Despite the critical role that identifiers play during program comprehension, there are no regulated naming standards for developers to follow when picking identifier names. Our research supports previous work aimed at understanding what makes a good identifier name, and practices to follow when picking names by exploring a phenomenon that occurs during identifier naming: identifier clones [3].

Identifier clones are two or more identifiers that are declared using the same name. This is an important yet unexplored phenomenon in identifier naming where developers intentionally give the same name to two or more identifiers in separate parts of a system. We must study identifier clones to understand it's impact on program comprehension and to better understand the nature of identifier naming. To accomplish this, we conducted an empirical study on identifier clones detected in open-source software engineered systems and propose a taxonomy of identifier clones containing categories that can explain why they are introduced into systems and whether they represent naming anti-patterns.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

During program maintenance, developers spend up to 70% of their time on program comprehension tasks [2]. Given that around 70% of source code characters are made up of identifiers [5], the majority of program comprehension tasks involve developers deriving meaning and intended behavior from the terminology found in identifiers. Identifiers are the main source of code documentation, and oftentimes they are the only source of documentation if there are no comments in the code being analyzed. This presents a problem as there is no standard way to name variables, as seen by a study by Fetirelson et al. [6] which shows there is a low probability that two developers will choose the same name for an identifier. Different developers are likely to use different terminology to represent the same concepts based on multiple factors including their background and exposure to the system at hand. Inconsistent identifier naming conventions hinder program comprehension. In order to work towards standardized models for naming identifiers, we must first understand what characteristics of identifiers improve comprehension and also understand what naming anti-patterns should be avoided. There have been multiple studies that look at how identifiers impact program comprehension [7–9], concluding that longer, more descriptive identifiers have a positive impact on comprehension. Some studies have shown that poor-quality identifier names have a direct negative impact on the readability of code [10]. Other studies have aimed to understand identifier structure by looking at their grammar patterns which can be used to automate the identifier naming process [11].

Our research explores a phenomenon in identifier naming that has yet to be explored: identifier clones. The term "identifier clones" refers to multiple identifiers that have been declared using the same name. To understand identifier clones and their impact on program comprehension we must first build a taxonomy of clones to understand what types of clones exist in the wild. Through an empirical study of identifier clones detected in software engineered open-source systems, we propose a taxonomy of identifier clones that can be used to classify clones based on their conciseness, consistency, origin, and behavior stereotypes. Our research supports the ongoing effort to better understand the nature of identifier naming [6, 12, 13], the characteristics that define high-quality identifier names [3], and the set of naming anti-patterns to be avoided when picking identifier names.

The remainder of this paper is as follows: Chapter 2 outlines the motivation of our research and research questions we set out to answer. Chapter 3 discusses related work. Chapter 4 details the selection of software systems containing identifier clones we analyzed, the research methodology we chose, and the infrastructure we built to support our empirical study. Chapter 5 discusses our findings in the form of our resulting taxonomy along with examples and distribution data. Chapter 6 discusses the threats to different validity concerns relevant to our study. And Chapter 8 concludes our research with a discussion of potential future work.

# Chapter 2

# Research Objective

## 2.1 Motivation

The goal of this paper is to investigate identifier clones found in the wild, understand their nature through static code analysis, and construct a taxonomy of identifier clones that can explain why they are introduced into software systems. As discussed in Chapter 3, other research studies have investigated the nature of identifier names and how they relate to program comprehension. However, there is no research exploring the phenomenon of identifier clones. Identifier clones is an interesting naming phenomenon to study since the action of developers declaring multiple identifiers using the same name can provide a new perspective in understanding naming patterns. Exploring identifier clones can also help us to discover new naming anti-patterns that can be used to improve identifier naming modeling used in automated identifier naming tools.

## 2.2 Contribution

Our primary contribution through our study is to enhance our understanding of identifier clones. We achieve this by building a taxonomy of identifier clones based on grounded observations made on real identifier clone occurrences seen in the wild.

## 2.3 Research Questions

In Grounded Theory, a research question may be defined prior to the study, which is usually open-ended in nature. Before starting our Grounded Theory research, we defined the open-ended research question of "Why are identifier clones introduced into software systems?". As we continued to analyze identifier clones, we refined our research questions based on the grounded observations and concepts that came up during our study. The following research questions help us answer what is the nature of identifier clones as seen in the wild:

- **RQ1:** ***Why are identifier clones introduced into software systems?*** Through empirical evidence observed on identifier clones seen in the wild, we propose a set of categories that characterize identifier clones and provide insight as to why they are introduced into systems. We do not claim to have found all categories and encourage further research in investigating this phenomenon.

- **RQ2:** ***What are the different factors that lead to the introduction of identifier clones in open-source software systems?*** Through relationships observed in source code functions containing identifier clones, we theorize factors that impact the introduction of identifier clones. We find that some factors are related to semantic relationships present in natural language (i.e. Homonymy) that are a source of ambiguity in identifier naming. A discussion of these factors can be seen in Chapter 5.

- **RQ3:** ***What is the resulting taxonomy categorizing identifier clones commonly found in open-source software systems?*** This question is concerned with understanding what are the final identifier clone categories we theorized in our study through the use of Grounded Theory. To better communicate the structure of our taxonomy, refer to Chapter 5.

- **RQ4:** ***What were the most common types of identifier clones? And why do these categories of clones show up frequently?*** Since we are interested in understanding the nature of identifier clones it is critical to understand the distribution of identifier clones analyzed in our study using our final categories. This helps us better understand what types of clones show up frequently and theorize why this is the case.

# Chapter 3

# Related Work

Several studies have looked at how to improve software maintenance in general [13–92]. More specifically, given that identifier names play a crucial role in code comprehension, there are several studies that aim to improve the quality of identifier names by defining characteristics of a high-quality name [3] and by exploring the nature of the identifier naming process [11–13]. Deissenbock et al. [3] define "Correctness", "Conciseness", and "Consistency" to characterize high quality identifier names. These concepts refer to whether the terminology in a name correctly describes the entity stored, how precisely the terminology represents an entity, and whether a name is consistently used throughout the system to represent the same entity. We used these concepts as a resource for conceptualizing, establishing relationships between the data we observed in our study, and generating identifier clone categories during coding and memoing activities of our Grounded Theory research study. We found that identifier clones may or may not be concise and consistent. Therefore, these concepts help in creating discrete logical groupings for identifier clones depending on whether the terminology used in identifiers is generic or precise, and whether the identifier name is consistently used to represent the same entity or not.

There are many research studies focusing on understanding the impact of identifier structure on program comprehension [7–9]. Schankin et al. [9] found that longer, more descriptive identifier names improve program comprehension. Their empirical study, which had a group of developers search for a semantic defect in a body of code, found that longer more descriptive identifier names resulted in the task being completed around 14% faster than when using shorter identifier names. Hofmeister et al. [7] conducted a similar study having a group of professional developers look for defects in source-code snippets and measured the time it took to perform this task when presented with identifiers written as full words, letters, or abbreviations. The authors found that using full words led to the task being completed 19% faster compared to when using letters and abbreviations. Lawrie et al. [8] investigate whether program comprehension is improved when identifiers include full words representing the concepts they represent. They conduct a study where participants are asked to describe functions containing common computer science algorithms (i.e. binary search) with the only difference being the use of full word identifiers versus their abbreviated versions. Their results also support that full word identifiers lead to better source code comprehension. These empirical studies all conclude that longer, more descriptive names improve program comprehension. We can reference these empirical studies for extending our research by constructing a similar study that measures whether identifier clones have an impact on program comprehension.

In addition, there are research studies focusing on improving our understanding of the nature of identifier naming. Newman et al. [11] investigate identifiers through studying grammar patterns with the goal of understanding identifier naming patterns that are used in supporting automated identifier naming tools. They do this by processing a large set of identifiers seen in the wild through a part of speech tagger that is able to tag each term composing an identifier. They conclude that current state-of-the-art part of speech taggers struggle to accurately tag the parts of speech on identifiers. This research also provides many insights as to how grammar behaves in identifiers seen in the wild. These grammar patterns are critical in understanding program semantics, as programmers use them to convey behavior in code. Peruma et al. [13] explore identifier rename refactorings on a large set of Java systems to understand why developers rename method, class, and package names in their code. One of their main research questions looked at how the semantic meaning of identifiers change as a result of a renaming refactoring. More specifically, they looked at whether the meaning of identifiers is broadened, narrowed, preserved or completely changed. This research uses a taxonomy of rename refactorings developed by Arnaoudova et al. [12] to tag identifier rename operations observed in the study. For example, the meaning is said to be modified if the meaning was generalized (i.e. old term renamed to a hypernym), or narrowed (i.e. old term renamed to a hyponym). In our research, we observe that the linguistic relationships that inspired the renaming categories used by Peruma et al. also plays a role in explaining why identifier clones are introduced into systems. For example, in our study we observed that hypernyms are a source of identifier clones as generic terms encapsulating a set of specific subtypes (i.e. "resource" is a hypernym of "autoScalingGroupResource") were used in any place where a subtype can be expected. This can be argued to hinder readability if a more concise name representing a specific subtype can be used instead. Our research therefore can add important information on whether a rename refactoring that generalizes or narrows the meaning of an identifier is a naming antipattern or not. Automated tools that perform rename refactorings that generalize the meaning of an identifier should take into account whether this refactoring will introduce identifier clones into the system and whether this will hinder code comprehension.

# Chapter 4

# Methodology

Given that no prior research has been done in exploring the nature of identifier clones, we carried out an inductive research approach to derive clone categories from grounded observations made on clones present in open-source systems. These clone categories aim to explain why identifier clones are introduced into a system and whether different types of identifier clones represent naming anti-patterns, or whether they can be explained by other factors that are to be expected in these types of systems (i.e. Hierarchical domain concepts). More specifically, we designed our research based on the Grounded Theory methodology. The following sections define in detail what steps were taken to select software systems for our study, what version of Grounded Theory was chosen, how we used GT to analyze source code as our primary source for codings, the template and database schema developed to generate and store codings, and examples of coding and memoing and how they impacted our final clone categories.

## 4.1 Selection of Software Systems

The pool of software systems publicly available to study identifier clones are endless thanks to repository hosting services like GitHub and Bitbucket. However, anyone can create repositories on these hosting services, creating noise for researchers to filter out when choosing software systems for their research projects. For example, Munaiah et al. [4] point out that some repositories do not represent quality software systems in the slightest, with some repositories being used to back up a computer's file system or representing throw away coding tutorials.

To avoid reaching inaccurate conclusions in our research, in the form of

Table 4.1: Software engineering dimensions and their corresponding practices and metrics taken into consideration by Muniah et al. [4] in determining software engineered systems

| **Dimension** | **SW Eng. Practice** | **Metric** |
|---|---|---|
| Community | Collaboration | Core Contributors |
| Continuous Integration | Quality | Uses CI Service |
| Documentation | Maintainability | Comment Ratio |
| History | Sustained Evolution | Commit Frequency |
| Issues | Project Management | GitHub Issue Frequency |
| License | Accountability | Contains License |
| Unit Testing | Quality | Test Ratio |

clone categories that are not reflective of practices you would find in software engineered projects, we employed a quality standard on how we picked the open-source software systems analyzed in this project. More specifically, we want to pick software systems that provide evidence that the developers involved in the development have made efforts to increase the quality of their system. A development team that does this will be more likely to spend time picking high quality identifier names during development, reducing the probability of encountering abnormal identifier naming behavior. Other research projects have used popularity measurements such as the number of GitHub stars a repository has, referred to as "GitHub Stargazers", to pick software systems for their research. This assumes that the popularity of a GitHub repository is correlated with the quality of the software project. However, as discussed in a study carried out by Munaiah et al. [4], the precision and recall of this strategy can be improved by also considering a set of software engineering practices commonly seen in high quality systems. The set of practices proposed can be seen in Table 4.1.

The set of practices chosen by Munaiah et al. provide evidence that a software system has followed software engineering practices including design, test, and maintenance. The evaluation framework developed by Munaiah et al. has some subjectivity, as pointed out by the authors, in terms of how they chose to measure and weight the different dimensions proposed to determine if a repository is a high-quality software engineered project or not. Despite these drawbacks, we chose to use their evaluation framework over other strategies since it provides a higher confidence that a software project has followed software engineering practices that would have an impact on the quality of identifier names present in a system.

Munaiah et al. developed a classifier called "reaper" that uses their evaluation framework to automate the classification of open-source repositories. To train this classifier, the authors manually created two sets of repositories containing software engineered projects[1]. The first set, named "Organization", contains repositories gathered from popular software engineering organizations (i.e. Netflix, Amazon, Google, etc.). The second set, named "Utility", are repositories that were deemed to provide a general-purpose utility to users. In addition to these criteria, the repositories chosen for each set must meet the criteria for a software engineered project based on their evaluation framework and software engineer dimensions. For our research project, we used these manually classified sets to pick software projects that scored highly in terms of Community, Documentation, History, and Unit Testing.

In addition to using these sets of software engineered projects to pick systems for our study, we also had the constraint of picking systems written in languages supported by the tool we used to detect identifier clones in source code. This tool, "IdentifierNameAndContext"[2], takes as input a srcML[3] archive (source code that has been compiled through srcML), which is used to scan over a repository and output detected clones along with the source code for the functions containing each declaration of an identifier clone. srcML is a free software application that compiles source code into XML format. This software application only supports the following languages: C, C++, C#, and Java. Given the development experience of the main researcher responsible for conducting codings, memoing, and conceptualization, we also decided to limit our study to systems written in Java. As discussed in Section 4.2, a core principle of Grounded Theory is "theoretical sensitivity", representing the ability for researchers to conceptualize given a set of data. Choosing languages unfamiliar to researchers would therefore hinder theoretical sensitivity. This decision presents a threat to the generalizability of our categories and can be used as a motivation for future work by analyzing identifier clones in other programming languages.

Following these process decisions, we selected 6 systems varying in size (lines of code), development team, and domain. This is to address theoretical gaps in our resulting clone categories that may be present in varying types of software systems as well as categories that may be a result of characteristics only found in specific types of systems. The software systems chosen can be

[1] https://gist.github.com/nuthanmunaiah/23dba27be17bbd0abc40079411dbf066

[2] https://github.com/SCANL/IdentifierNameAndContext

[3] https://www.srcml.org/#home

Table 4.2: Software Systems chosen for our study along with metrics scoring their software engineering dimensions

| Repository | Size | Contributors | Documentation | Testing | Github Stars |
|---|---|---|---|---|---|
| Stash-hook-mirror | 989 LOC | 1 | 0.12 | 0.51 | 32 |
| SimianArmy | 16,086 LOC | 8 | 0.33 | 0.32 | 3,717 |
| Sqoop | 75,520 LOC | 5 | 0.27 | 0.26 | 172 |
| Maven | 103,384 LOC | 7 | 0.26 | 0.07 | 436 |
| Phoenix | 182,447 LOC | 9 | 0.19 | 0.38 | 158 |
| Activemq | 391,731 LOC | 8 | 0.24 | 0.35 | 419 |

seen in Table 4.2.

After identifying the software engineered systems to include in our study, we then continued to use the tool “IdentifierNameAndContext” to detect the population of clones present in the chosen systems. We then applied "Stratified Sampling" to the population of clones with the goal of reducing the manual effort of analyzing thousands of identifier clone instances. The populations and samples of clones to be analyzed for each system can be seen in Table 4.3, along with the number of codings to be performed. This is considering that each identifier clone instance (each declaration of a variable sharing the cloned name) will require a coding. You can find the raw text files containing all detected clones, samples of clones, and source code for functions containing detected clones in our GitHub repository [4].

The Stratified Sampling performed on populations of clones follows the following steps:

1. Place identifier clones into subpopulations based on their frequency (number of times the name was used in declaring a variable)

2. Iterate over subpopulations, picking a random identifier clone from each subpopulation

3. Repeat previous step until we reach 95% CI Sample

The Python script performing this sampling can be seen in our GitHub repository [5]

[4] https://github.com/SCANL/identifier_clones_GT_project/tree/main/Repositories

[5] https://github.com/SCANL/identifier_clones_GT_project/blob/main/StratifiedSampling/stratifiedsampling.py

Table 4.3: Detected identifier clone populations and samples for chosen software engineered systems as well as number of codings to perform to estimate effort

| **Repository** | **Clone Population** | **# of Codings** | **95% CI Sample** | **# of Codings** |
|---|---|---|---|---|
| Stash-hook-mirror | 20 | 60 | NA | NA |
| SimianArmy | 253 | 1,594 | 153 | 1,372 |
| Sqoop | 717 | 5,621 | 250 | 4,310 |
| Maven | 891 | 8,788 | 269 | 6,966 |
| Phoenix | 2,610 | 27,079 | 335 | 18,057 |
| Activemq | 1,911 | 41,315 | 320 | 34,826 |

Given the time constraint for this research study, we were only able to analyze clones present in Stash-hook-mirror, and SimianArmy. This was primarily due to the large manual effort involved in performing the codings and memos. With an average of completing a coding every four minutes, recording a memo for a clone in around five minutes, and checking if the new data generates a new category taking around 5 minutes, this effort for "stash-hook-mirror" and "SimianArmy" alone takes around 125 hours to complete. With codings being the most time-consuming activity, taking around 95.5 hours. Only analyzing identifier clones in two systems is a threat to the validity of our study as it is possible that theoretical saturation was not reached given that we did not sample clones from additional software systems varying in size, development team, and domain, which are system characteristics that could impact the introduction of identifier clones.

## 4.2 Grounded Theory

Grounded Theory is a process methodology that follows the inductive paradigm for generating theories grounded in empirical evidence, called “codings”. The reported theory for a research paper following this process methodology may be in the form of a conceptual framework, conceptual mode, set of factors, or set of themes or categories that provide an explanation for a certain behavior [93]. For this thesis, the proposed theory is in the form of a taxonomy of identifier clones, where each category provides an insight as to why clones were introduced into a software system. Prior to starting data collection and analysis, we designed the process to carry out our study based on GT’s core principles and best practices as outlined by Stol et al. The following are initial considerations and process decisions we discussed abiding by GT's core principles:

- **Limit exposure to literature.** To avoid bias in the process of analyzing detected clones, gathering grounded observations, and deriving theories for why clones are introduced into a system, we did not do a prior literature review and limited our use of literature. However, we did reference related literature on identifier naming to improve our ability to conceptualize and form relationships between data analyzed. This is a viable strategy in Straussian GT.

- **Treat everything as data.** The tool we used to detect identifier clones present in software systems outputs the source code for function blocks in which clones are declared. When performing analysis on this free form data (source code), we did not put any constraints on the type of observations that can be made. Instead, we created a form template for researchers to perform codings and annotate their observations. The form template provides a combination of free-form inputs (i.e. how is the clone being used in the containing function) and boolean type inputs (i.e. was the clone declared as a method parameter). The form can be seen in Figure 4.2. This structure for collecting codings helped in providing a starting set of generic clone characteristics to observe but also did not restrict our codings to a limited set of observation types.

- **Immediate and continuous data analysis.** We performed coding and memoing simultaneously, and recorded the progression of these activities on our GitHub repository. Codings were captured in markdown files [6] versioned by the date in which it was generated. Memos were captured in Microsoft Word documents [7] also versioned by date. More details on the coding and memoing activities can be seen in Section **??**.

- **Theoretical sampling.** Although there is a finite population of clones in each open-source software system, theoretical sampling can be done by choosing additional software systems to fill in gaps in clone categories defined from observing previous systems. Additional systems can be continued to be added for analysis until theoretical gaps are saturated. At the start of the project, we selected six systems to analyze with the

[6] `https://github.com/SCANL/identifier_clones_GT_project/tree/main/MarkdownFiles/IdentClonesCodingsFiles`

[7] `https://github.com/SCANL/identifier_clones_GT_project/tree/main/MemosNotes`

goal of reducing theoretical gaps. However, due to time constraints, we only analyzed the clones present in two software systems.

- **Theoretical sensitivity.** We established weekly meetings to discuss codings, related concepts between codings, and how clone categories (theories) are impacted by any new data encountered through coding. This was performed by two researchers to improve the theoretical sensitivity. We also referenced related works on identifier naming [3, 94] to improve our ability to derive relationships between the data analyzed and conceptualize new categories of identifier clones.

- **Coding.** Codings were performed through static code analysis of function blocks containing each individual identifier clone instance detected. For example, when analyzing a new identifier clone that was declared eight times across a software system, we record grounded observations for each identifier declared that shared the clone name.

- **Memoing.** Memos were recorded for every new clone observed, documenting how new data relates to the current set of theories (clone categories) and performing any updates on those theories if necessary.

- **Constant comparison.** Each new data point (identifier clones) was compared against previous observations made on past data points to establish relationships between the clones observed and generate categories. This often required us to update the ongoing clone categories at the start of the project. The process of relating new data points to past observations and categories was done on a weekly basis.

- **Cohesive theory.** Transitioning from the ongoing categories emerging from the analysis of new clones, we developed a set of cohesive categories that combined all emerging categories. Our final theory was a set of decision trees that can be used to classify new clones based on different characteristics including their conciseness, consistency, origin, and generic behavior. The final set of categories is able to categorize all the clones observed in this study.

- **Theoretical saturation.** We reached theoretical saturation for the software systems included in this study. All clones found in these systems fit into the final categories proposed. However, this research will be extended to observe 4 additional software systems that vary in domain, size, and development team to have a stronger argument for generalizability of our categories. Future work can be done by analyzing identifier clones in systems programmed in different languages as well to remove this as a potential theoretical gap in our findings.

As discussed by Stol et al. [6], there are three main versions of Grounded Theory consisting of Glaserian GT, Straussian GT, and Constructivist GT. One of the main differences being the process of deriving theory. Some versions are more faithful to the data, meaning that any derived category must be purely based on concrete data observed during the study (Classical GT). While other versions allow for a more flexible process for deriving theory (Straussian GT). Given that this is the first-time identifier clones are being explored we picked Straussian GT since it is more flexible on conceptualizing relationships in the data and allowed us to investigate additional sources for theories such as Linguistic Relationships or related literature on identifier naming [3, 94]. For example, a set of categories we propose classify clones based on the clone origin or resource a developer must reference to interpret the correct meaning of an identifier. These categories are not purely based on the data we collected in our study, since we did not analyze the set of resources where you can find the correct meaning for each identifier (Project Requirements, Developer Terminology, English Dictionary). Despite these data sources not being included in our research study, we were still able to theorize that the origin of the meaning for an identifier has an impact on whether identifier clones are introduced into a system. This is an acceptable practice in Straussian GT.

## 4.3 Deviations from Straussian Grounded Theory

As Stol et al. [93] point out, Grounded Theory coding and data analysis practices were developed for unstructured text which have been primarily used to analyze data in fields unrelated to Software Engineering. To analyze source code, we took the advice of Stol et al. and employed static code analysis on the source code of functions containing identifier clone instances. Therefore, we did not use the conditional matrix for coding, which is mentioned in the Straussian GT version.

Grounded Theory also uses "Theoretical Sampling" instead of conventional sampling techniques. We argue that "Theoretical Sampling" is performed in our study by including additional systems with varying characteristics until theoretical saturation is reached. Which is something to be extended for future research. For each chosen system, we can perform a conventional sampling technique to reduce the manual effort of making grounded observations on thousands of identifiers. We used "Stratified Sampling" to achieve this, reducing the number of identifiers analyzed while maintaining a 95% confidence level on representing the characteristics of the population of clones in a system.

## 4.4 Study Infrastructure

After choosing the systems to be included in our study, we continued to develop the infrastructure to support tracking the large number of data points to be analyzed. Given that we wanted control over querying the grounded observations recorded for each identifier clone, we built a MySQL database reflecting the Entity-Relationship diagram in Image 4.1. As can be seen by the crow's foot notation used in the diagram, the clones stored in the "clones_data" table are connected to one or many observations stored in the "clones_observations" table. Each entry in the "clones_observations" table represents a series of grounded observations collected for a single identifier clone instance. For example, if the identifier clone "resource" is declared in eight different places in a system, then we record eight separate observations in "clones_observations" table for each time the identifier clone name was used in the declaration of a


Figure 4.1: A picture of the Entity Relationship Diagram representing our database storing codings and memos

variable. Although we did not have time to insert our memos recorded using MS Word into our database, we propose a schema that supports a hierarchical structure of memos. This schema was inspired from our goal of having Ground Level Memos, 1st Level Memos, 2nd Level Memos, and so on, where each subsequent level gets closer to a finalized theory. Since a memo can either summarize the findings of a set of observations or a set of lower level memos, we built two separate junction/join tables called "parentmemo_to_childmemo" and "observations_to_memos" to support these relationships.

In order to facilitate the effort performing GT codings, we built a lightweight UI application using React.js and Express.js, which can be run using the Node.js runtime environment. The code can be found in our GitHub repository [8], which is broken down into two modules: client, and server. The client module represents the React.js application containing the UI form used by researchers in our study to record new codings on identifier clone instances. The server module represents the backend application built on Express.js (Node.js web application framework) used to connect to a local database instance reflecting our schema from Image 4.1. A screenshot of the UI form used to perform the codings can be seen in Image 4.2. Some inputs were omitted from this screenshot to save space. The full view can be seen by cloning our repository and running our client and server applications.

As can be seen on our Codings UI form, we follow a static code analysis approach to perform the codings. This is to say that we record syntactic and code structure information relating to the identifier clone being observed. For example, recording whether an identifier was included in any looping structures or the return statement of a function block. These inputs are in the form of checkboxes indicating whether this is true or false for the identifier clone instance being analyzed. We also provide free-form inputs (i.e. "Method Behavior Summary") that researchers used to input a brief summary of the behavior of the function being observed as well as how the identifier clone is being used in the function.

In addition to building the UI form application to insert codings, and memos into a MySQL database reflecting our schema, we also built a Python script that queries the codings in the database and generates a readable Markdown[9] file to analyze ongoing codings. This facilitated the process of conceptualizing, forming relationships between clones observed, and generating new categories. Our versioned markdown files can be seen in our repository[10]. The

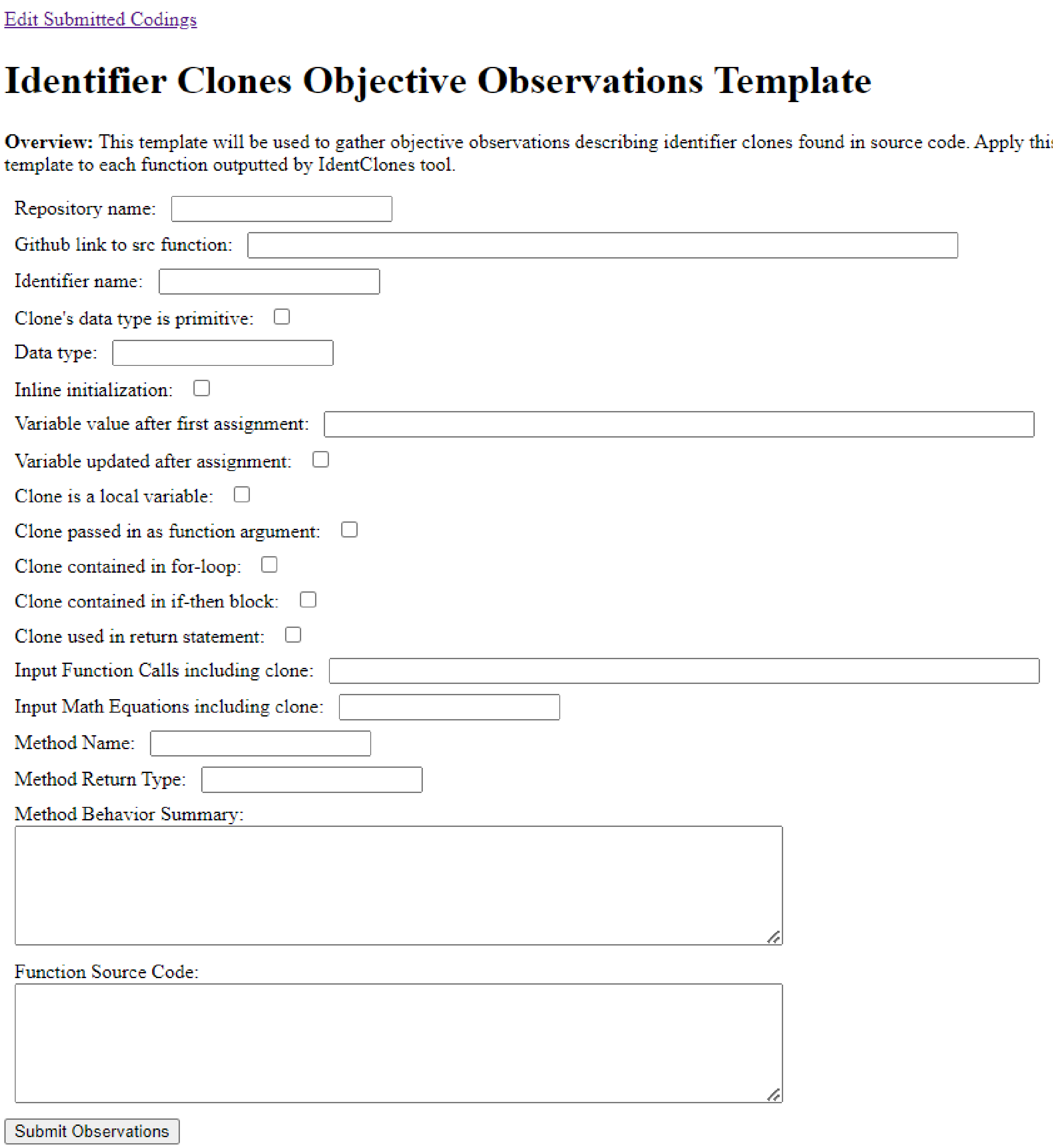
[8] https://github.com/SCANL/IdentifierClonesObservationsApp

[9] https://www.markdownguide.org/getting-started/

[10] https://github.com/SCANL/identifier_clones_GT_project/tree/main/MarkdownFiles/IdentClonesCodingsFiles

Python script to generate the Markdown files is stored in our repository as well[11]

[11] https://github.com/SCANL/identifier_clones_GT_project/blob/main/MarkdownGeneratorScripts/IdentClonesMarkdownGenerator.py



Edit Submitted Codings

# Identifier Clones Objective Observations Template

**Overview:** This template will be used to gather objective observations describing identifier clones found in source code. Apply this template to each function outputted by IdentClones tool.

Repository name:

Github link to src function:

Identifier name:

Clone's data type is primitive:

Data type:

Inline initialization:

Variable value after first assignment:

Variable updated after assignment:

Clone is a local variable:

Clone passed in as function argument:

Clone contained in for-loop:

Clone contained in if-then block:

Clone used in return statement:

Input Function Calls including clone:

Input Math Equations including clone:

Method Name:

Method Return Type:

Method Behavior Summary:

Function Source Code:

Submit Observations

Figure 4.2: A picture of the UI Form used to perform the codings (grounded observations) on identifier clone instances

# Chapter 5

# Analysis & Discussion

In this chapter, we list our proposed identifier clone categories and provide examples of codings and memos that led to the conceptualization of our final categories. We propose four non-mutually exclusive sets of categories: "Traveling Clones", "Clone Consistency", "Clone Origin", and "Identifier Behavior Stereotypes". These sets of categories are able to characterize an identifier clone in terms of different dimensions, providing insight as to why the identifier was cloned and whether it represents a naming anti-pattern. In addition, we report the frequency of identifier clones that fall under each category, with exception of the "Identifier Behavior Stereotypes" categories, as we did not have time to go back and label the behavior stereotypes of each declaration of a cloned variable.

## 5.1 Conciseness

The first set of categories proposed deal with how precisely the terminology used in an identifier clone name represents the entity being stored while using as few words as possible. These categories were inspired by Deissenboeack et al.'s definition of "Conciseness" [3], which classifies an identifier as concise if and only if the identifier name exactly matches the concept name of the entity represented by the identifier. Therefore, in their definition of conciseness, the number of terms needed for an identifier to be concise will be determined by the size of the concepts included in a system. We propose three categories as seen in Figure 5.1 to describe the conciseness of an identifier name: Generic Identifiers, Specific Identifiers, and Imprecise Identifiers. Classifying identifier clones into these conciseness categories can be done by determining whether the meaning of the identifier name is correct when applied to all contexts in a

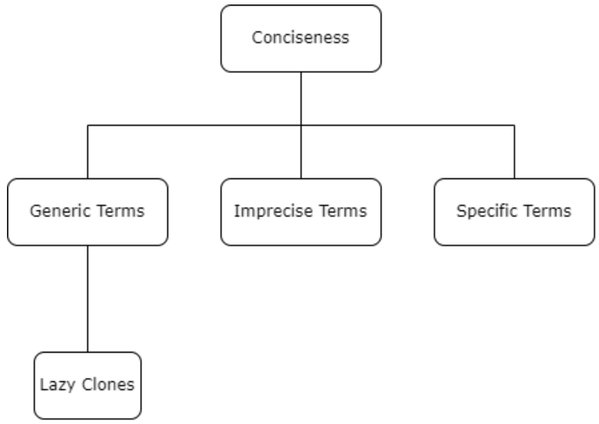



Figure 5.1: Identifier Clone Conciseness Categories Diagram

system, multiple contexts, or is only correct in a single context. This is similar to the "conciseness" definition by Deissenboeck et al. [3], where they define an identifier name is concise if and only if the identifier name is exactly the same name as the concept it represents. However, we measure conciseness on whether the meaning of an identifier name becomes incorrect when placed in different contexts in a system.

### 5.1.1 Generic Identifiers

Identifier clones classified as Generic Identifiers use abstract terminology that can be applied to any context in a system. Given that this requires the meaning of a name to be correct in any context, the number of encountered identifier clones that fall under this category was small. We only classified 5 identifier clones as Generic during our study. These clones are listed in Table 5.1.

Table 5.1: Identifier Clones classified as Generic Identifiers

| **Identifier** | **Repository** | **Conciseness** | **Consistency** | **Origin** |
|---|---|---|---|---|
| values | stash-hook-mirror | Generic | Domestic Traveling | Natural Language |
| value | SimianArmy | Generic | Non-Traveling | Natural Language |
| n | SimianArmy | Generic | Non-Traveling | NA |
| elem | SimianArmy | Generic | Non-Domestic Traveling | Natural Language |
| data | SimianArmy | Generic | Domestic Traveling | Natural Language |

**Generic Identifier** Example #1 "value" was detected in the SimianArmy repository. This identifier was cloned 26 times. 20 of these variable declarations had the data type "String", 1 had data type of "Enum⟨?⟩", 1 had data type of "NamedType", 2 had data types of "Date", and 2 had data types of "boolean". This identifier name was used in multiple contexts, including de-serializing the key-value pairs in a map data structure into an internal data object (i.e. AWSResource), converting encoded strings into enums, and de-serializing json objects into internal Resource objects. This is an example of an identifier that is both Generic and Non-Traveling (not used consistently). Listings 5.1 and 5.2 represent two instances of the identifier clone declared in methods "parseJsonElementToresource" and "valueToEnum".

```
private Resource parseJsonElementToresource(String region, JsonNode jsonNode
        , Map<String, Long> lcNameToCreationTime) {
        Validate.notNull(jsonNode);

        String asgName = jsonNode.get("autoScalingGroupName").getTextValue();
        long createdTime = jsonNode.get("createdTime").getLongValue();

        Resource resource = new AWSResource().withId(asgName).withRegion(region)
                .withResourceType(AWSResourceType.ASG)
                .withLaunchTime(new Date(createdTime));

        JsonNode tags = jsonNode.get("tags");
        if (tags == null || !tags.isArray() || tags.size() == 0) {
            LOGGER.debug(String.format("No tags is found for %s",
                resource.getId()));
        } else {
            for (Iterator<JsonNode> it = tags.getElements(); it.hasNext();) {
                JsonNode tag = it.next();
                String key = tag.get("key").getTextValue();
                String value = tag.get("value").getTextValue();
                resource.setTag(key, value);
            }
        }

        ...

        return resource;

    }
```

Listing 5.1: Generic Identifier "value" detected in SimianArmy. This function deserializes a json object into an internal Resource object. Clone "value" is used to set tag field on newly constructed Resource object.

```
    /**
     * Value to enum. Converts a "name|type" string back to an enum.
     *
     * @param value
     *            the value
     * @return the enum
     */
    public static <T extends NamedType> T valueToEnum(
            Class<T> type, String value) {
        // parts = [enum value, enum class type]
        String[] parts = value.split("\\|", 2);

        ...

        @SuppressWarnings("rawtypes")
        Class<? extends Enum> enumType = enumClass.asSubclass(Enum.class);
        @SuppressWarnings("unchecked")
        T enumValue = (T) Enum.valueOf(enumType, parts[0]);
        return enumValue;
```

```
}
```

Listing 5.2: Generic Identifier "value" detected in SimianArmy. This function converts an encoded string into an Enum. Clone "value" represents the encoded string.

**Generic Identifier** Example #2 "data" was detected in the SimianArmy repository. This identifier was cloned 4 times. All variables were declared using the same data type "JsonNode" and represent the same entity being a json element contained in the data fetched from an external data source. This is an example of an identifier that is both Generic and Domestic Traveling (used consistently). Listings 5.3 and 5.4 represent two instances of the identifier clone declared in methods "addLastAttachmentInfo" and "refreshId-ToCreationTime".

```
/**
 * Adds information of last attachment to the resources.
 * @param resources the volume resources
 */
private void addLastAttachmentInfo(List<Resource> resources) {
    LOGGER.info(String.format("Updating the latest attachment info for %d
         resources", resources.size()));
    ...
    for (Map.Entry<String, List<Resource>> entry :
         regionToResources.entrySet()) {
        for (List<Resource> batch : Lists.partition(entry.getValue(),
             BATCH_SIZE)) {
            String batchUrl = getBatchUrl(entry.getKey(), batch);
            JsonNode batchResult = null;
            batchResult = eddaClient.getJsonNodeFromUrl(batchUrl);

            Set<String> processedIds = Sets.newHashSet();
            for (Iterator<JsonNode> it = batchResult.getElements();
                 it.hasNext();) {
                JsonNode elem = it.next();
                JsonNode data = elem.get("data");
                String volumeId = data.get("volumeId").getTextValue();
                Resource resource = idToResource.get(volumeId);
                JsonNode attachments = data.get("attachments");

                ...
                processedIds.add(volumeId);
                setAttachmentInfo(volumeId, attachment, detachTime, resource);
            }

            ...
        }
    }
}
```

Listing 5.3: Generic Identifier "data" detected in SimianArmy. This function updates the "last attachment information" field on list of Resource objects. Clone "data" represents the last attachment information fetched from external AWS Edda Service.

```
/**
 * AWS doesn't provide creation time for images. We use the ctime (the creation
       time of the image record in Edda)
 * to approximate the creation time of the image.
 */
private void refreshIdToCreationTime() {
```

```
        for (String region : regions) {
            String url = eddaClient.getBaseUrl(region) + "/aws/images";
            LOGGER.info(String.format("Getting the creation time for all AMIs in
                 region %s", region));
            url += ";_expand;_meta:(ctime,data:(imageId))";

            JsonNode jsonNode = eddaClient.getJsonNodeFromUrl(url);

            ...

            for (Iterator<JsonNode> it = jsonNode.getElements(); it.hasNext();) {
                JsonNode elem = it.next();
                JsonNode data = elem.get("data");
                String imageId = data.get("imageId").getTextValue();
                JsonNode ctimeNode = elem.get("ctime");
                if (ctimeNode != null && !ctimeNode.isNull()) {
                    long ctime = ctimeNode.asLong();
                    LOGGER.debug(String.format("The image record of %s was created
                         in Edda at %s",
                            imageId, new DateTime(ctime)));
                    imageIdToCreationTime.put(imageId, ctime);
                }
            }
        }
        LOGGER.info(String.format("Got creation time for %d images",
             imageIdToCreationTime.size()));
    }
```

Listing 5.4: Generic Identifier "data" detected in SimianArmy. This function updates the "Creation Time" values for AWS images stored in class data member map "imageIdToCreationTime". Clone "data" represents the image information fetched from external AWS Edda Service.

### 5.1.2 Specific Identifiers

Identifier clones classified as Specific Identifiers use precise terminology that can be applied to only one context in a system. If we try to place a specific identifier in any other context within the system, its meaning will be incorrect for that given context. For example, the meaning of the clone name "exludedImageIds", detected in the SimianArmy repository, is only correct in one context within the system, which is a collection of aws image ids that have been excluded from being some process. We detected 50 Specific Identifiers during our study. You can view a small list of examples in Table 5.2

### 5.1.3 Imprecise Identifiers

Identifier clones classified as Imprecise Identifiers use terminology that can be applied to multiple, but not all, contexts in a system. In other words, there is a set of contexts within the system on which we can place an imprecise identifier and its meaning will remain correct. For example, the meaning of the clone name "list", detected in the SimianArmy repository, is correct in any context within the system where a list of elements is expected. However, given that the identifier name does not provide information as to what elements are stored

Table 5.2: Identifier Clones classified as Specific Identifiers

| Identifier | Repository | Conciseness | Consistency | Origin |
|---|---|---|---|---|
| excludedImageIds | SimianArmy | Specific | Domestic Traveling | Project + NL |
| dnsEntryList | SimianArmy | Specific | Non-Domestic Traveling | Developer + NL |
| lcName | SimianArmy | Specific | Non-Domestic Traveling | Project + NL |
| lcNameToCreationTime | SimianArmy | Specific | DomesticTraveling | Project + NL |
| lcCreationTime | SimianArmy | Specific | Non-Domestic Traveling | Project + NL |
| elbClient | SimianArmy | Specific | Domestic Traveling | Developer + Project |
| volumeIds | SimianArmy | Specific | Non-Domestic Traveling | Project + NL |
| dnsType | SimianArmy | Specific | Non-Domestic Traveling | Developer + NL |
| monkeyType | SimianArmy | Specific | Domestic Traveling | Project + NL |
| resourceRegion | SimianArmy | Specific | Domestic Traveling | Project |

in the list, this name is not precise and the meaning is correct when applied to multiple contexts. In practice we have noticed that there is a range of how concise an identifier can be. However, in our research we are not measuring how concise an identifier is if they fall under the Imprecise Identifiers category. We are only differentiating imprecise identifiers from precise and generic identifiers. We detected 114 Imprecise Identifiers during our study. You can view a small list of examples in Table 5.3

Table 5.3: Identifier Clones classified as Imprecise Identifiers

| Identifier | Repository | Conciseness | Consistency | Origin |
|---|---|---|---|---|
| encryptedData | stash-hook-mirror | Imprecise | Non-Domestic Traveling | Developer |
| errors | stash-hook-mirror | Imprecise | Domestic Traveling | Natural Language |
| request | stash-hook-mirror | Imprecise | Non-Traveling | Natural Language |
| client | SimianArmy | Imprecise | Non-Traveling | Developer |
| result | SimianArmy | Imprecise | Non-Traveling | Natural Language |
| query | SimianArmy | Imprecise | Non-Domestic Traveling | Developer |
| request | SimianArmy | Imprecise | Non-Domestic Traveling | Developer |
| resource | SimianArmy | Imprecise | Non-Domestic Traveling | Project |
| input | SimianArmy | Imprecise | Non-Traveling | Natural Language |
| id | SimianArmy | Imprecise | Non-Traveling | Natural Language |

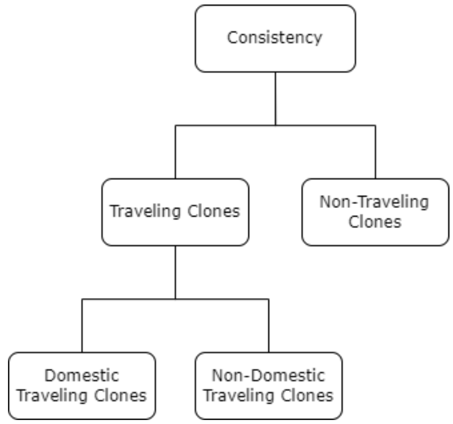



Figure 5.2: Identifier Clone Consistency Categories Diagram

## 5.2 Traveling Clones

The "Traveling Clones" categories deal with whether an identifier clone is used consistently across a system. An identifier clone is said to be used consistently if all identifiers sharing the cloned name represent the same entity and are used in the same way in the scope in which they are declared. For identifier clones classified as "Traveling" (used consistently), we also define the categories "Domestic Traveling Clones" and "Non-Domestic Traveling Clones" that describe the spread of a clone throughout a system. By "spread" we mean are all instances of a clone declared in a single file, set of cohesive files sharing behavior, or are declared across multiple files unrelated to the behavior they provide to the system. We use the term "Domestic" to describe clones that are co-located in the same file or in a set of cohesive files. Other traveling clones are classified as "Non-Domestic".

The measurement we used to determine if an identifier is consistent or not involved a series of checks:

- Do all the identifier clone instances share the same data type or a similar data type, where "similar" means data types that share a common behavior (i.e., data types "List" and "ArrayList" are similar data types in that they share the common behavior of storing a collection of items)
- Do all the identifier clone instances share a cohesive set of generic Identifier Behavior Stereotypes (Discussed in Subsection 5.4)
- Do all the identifier clone instances appear to perform the same behavior

in code as observed through static code analysis during the GT coding activity performed in our study.

## 5.3 Identifier Origin

The "Identifier Origin" categories deal with determining what resource, or origin, a developer must refer to in order to understand the correct meaning of an identifier name. This is an important category to understand whether ambiguous terminology in natural language stemming from semantic relationships such as homonyms (words spelled the same having multiple meanings) leads to the introduction of identifier clones. This category was created from conceptualizing that the more context provided in the terms used in an identifier name, the less likely it will result in a clone that is generic and used inconsistently. For example, if an identifier uses terminology from Developer Terminology, Project Domain, and Natural Language, then it will have a higher probability of being more precise in representing the entity stored since it uses context from various sources. This theory was reinforced by our findings for Generic clones in which we found that the only five clones detected as "Generic" all had the Origin category "Natural Language" (In exception for the clone "n" which does not have an Origin since "n" just acts as a placeholder name).

The process in which we classify the clone origin of an identifier clone involves first splitting the identifier into its atomic words following the camel casing naming format (individual terms composing the identifier). Then, for each atomic word we determine whether the correct meaning comes from Natural Language (i.e. English Dictionary), Developer Terminology, or Project Requirements/Domain. For example, the identifier clone "trackedMarkedResources" is split into the atomic words: "tracked", "Marked", and "Resources". Then we observe that both "tracked" and "Marked" are Natural Language terminology, and "Resources" is Project Domain terminology ("Resources" refers specifically to AWS resources). In addition, if the atomic words in an identifier clone are abbreviations, we first expand them before determining their origin. For example, the identifier clone "asgList" is first broken into the atomic words: "asg" and "List". "asg" is first expanded to "Auto Scaling Group". Then we observe that "Auto Scaling Group" is Project Domain terminology and "List" is Developer terminology.

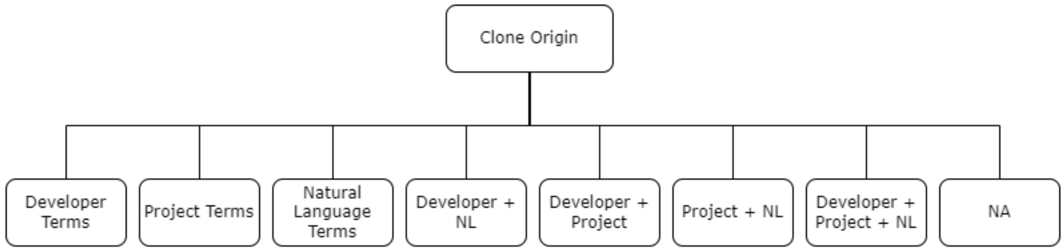



Figure 5.3: Identifier Clone Origin Categories Diagram

## 5.4 Identifier Behavior Stereotypes

The "Identifier Behavior Stereotypes" consists of categories that represent abstract generic behaviors that a variable in code can perform. For example, a variable that is used in the process of iterating over some collection of elements (i.e., pointer, or looping index variable) falls under the "Iterator" category. Another example is a variable that is used in the evaluation of a boolean expression is classified as "Predicate". The idea of creating "Behavior Stereotypes" to summarize the behavior of identifier clones to measure consistency was taken from Method Stereotypes used by [94]. The goal of these categories was to support our measurement for determining whether identifier clones are used consistently or not. If we notice that all variables sharing a cloned name have the same generic behavior, then this supports the argument that an identifier clone is used consistently. The following are descriptions for each Identifier Behavior Stereotype we propose.

### 5.4.1 Accessor

Identifier is used to fetch data (could be internal or external to the system). Subtypes of the "Accessor" category:

1. Structural Accessor: Identifier queries the state of an internal object.
   (a) Structural Accessor Property: Identifier stores the state of an object to be used in a method.
   (b) Structural Accessor Modifier: Identifier modifies how state of an object is fetched. For example, a variable that is passed as a method argument in a "getter" method call.
2. External Data Accessor: Identifier queries data from an external data store.

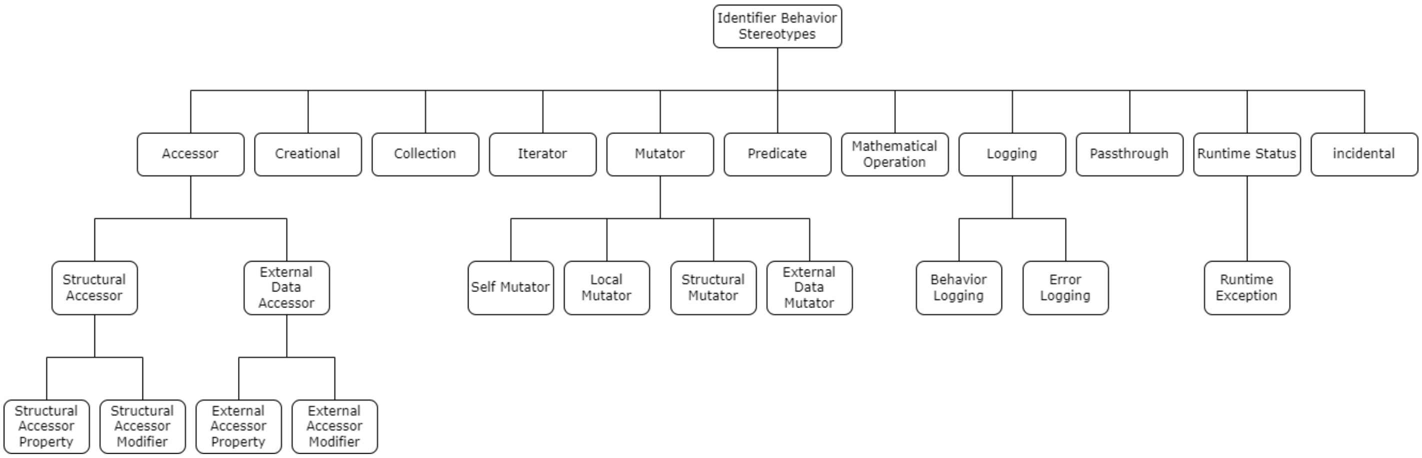



Figure 5.4: Identifier Clone Behavior Categories Diagram

(a) External Accessor Property: Identifier stores the state of data fetched from external data store.

(b) External Accessor Modifier: Identifier modifies how external data is fetches. For example, a variable that serves as a query filter parameter in a SELECT database query.

### 5.4.2 Creational

Identifier is used in the construction of new objects. For example, a variable that is passed into the constructor of a new object.

### 5.4.3 Collection

Identifier stores multiple data elements accessed or mutated in the containing function. Functions commonly iterate over elements in the variable or update elements inside.

### 5.4.4 Iterator

Identifier is used to iterate over items in a collection. Commonly declared within a programming looping structure (e.g., for-loop, while loop, etc.).

### 5.4.5 Mutator

Identifier is used to update data (could be internal or external to the system).

1. Self Mutator: Identifier state is updated within the method after being declared and initialized
2. Local Mutator: Identifier is used to update data value of another local variable. Where a local variable is declared within the same function or is a data member of the same class instance or is a static data member of the same class.
3. Structural Mutator: Identifier is used to update the state of an internal object. For example, a variable that is passed as a method argument to a "setter" method called on the object that it is modifying.
4. External Data Mutator: Identifier is used to update external data. For example, a variable that is passed as argument to an api definition of an external service that is documented to perform a state update.

### 5.4.6 Predicate

Identifier is used in the evaluation of a boolean expression. Commonly used in conditional statements or used in the return statement of a predicate method. For example, a variable that is used in the evaluation of a boolean expression inside a conditional statement (i.e., if, if-else, or switch statements).

### 5.4.7 Mathematical Operation

Identifier is used in the calculation of a mathematical expression.

### 5.4.8 Logging

Identifier is used to log information.

1. Behavior Logging: Identifier is used in logging normal method behavior
2. Error Logging: Identifier is used in logging errors in method. Often found inside catch blocks.

### 5.4.9 Passthrough

Identifier is not used inside the function in which it is declared but is passed as a function argument to another function call.

### 5.4.10 Runtime Status

Identifier is used in the process of providing visibility to the runtime status. For example, the identifier "responseStatus" in SimianArmy informs the developer whether a POST API request resulted in an error or success.

1. Runtime Exception: Identifier is used in the process of throwing a runtime exception

### 5.4.11 Incidental

Identifier is declared but not used.

## 5.5 Identifier Clone Categories Frequency

The Pie Charts depicted in Figures 5.5, 5.3, and 5.4 summarize the distribution of detected clones in our study on our proposed categories. The frequencies are the result of a manual effort on labeling the 169 observed clones in our study

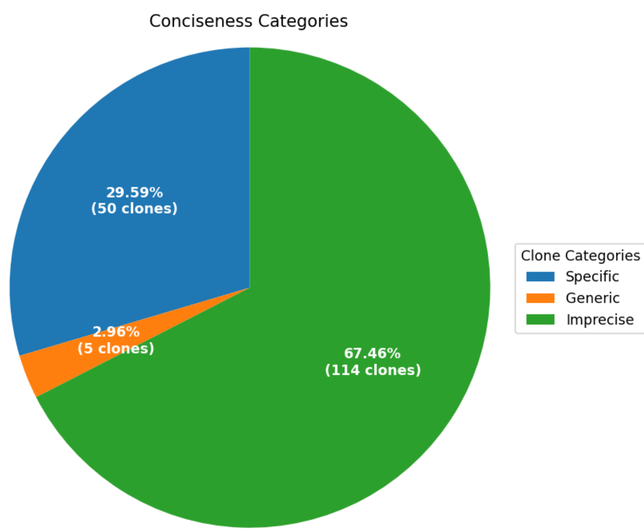



Figure 5.5: Identifier Clone Conciseness Categories Frequency Pie Chart

### 5.5.1 Categories Frequency Observations

- Only 3% of clones detected fall under the Conciseness "Generic" category. Meaning that it was rare for the naming of an identifier to apply to all

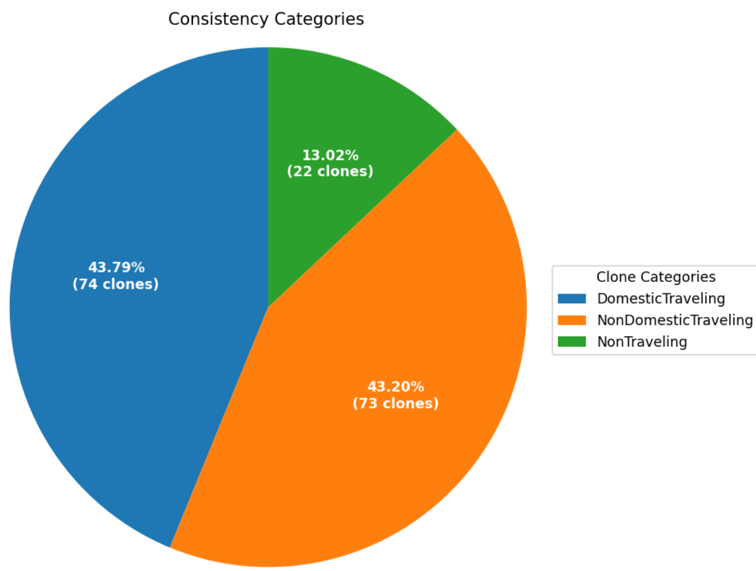



Figure 5.6: Identifier Clone Consistency Categories Frequency Pie Chart

contexts in a system. This may be due to the higher quality of the systems analyzed where developers avoid using generic lazy terms such as "value". This could also be a factor of generic terminology being rare across all naming in software engineered open-source systems. Additional work would be needed to determine whether this observation can be generalizable to other systems.

- A large portion (42%) of identifier clones observed were classified as having a clone origin of "Natural Language". Both clone origins "Project" and "Developer" still had significant portions with 14% and 15% respectively. The clone origins that combined terminology from different sources were seen significantly less often, except for "Project + Natural Language". This may be a sign that terminology from different sources provide more contextual information in an identifier name which in turn, reduces the likelihood of an identifier name being introduced into a system. However, we do not have sufficient data to claim there is a strong causation relationship between the origin of the terminology in an identifier and identifier clones being introduced into a system.

- Only 13% of clones detected fall under the Consistency "Non-Traveling" category (not used consistently). This means that the majority of clones

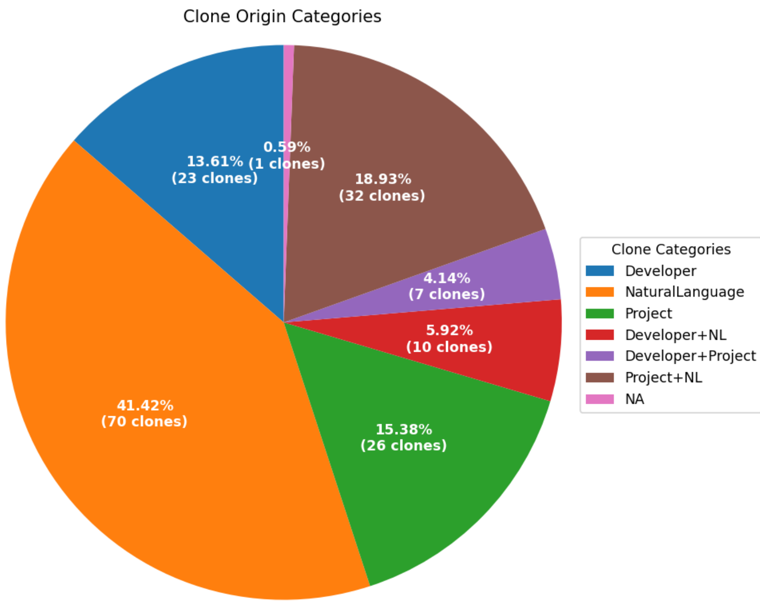



Figure 5.7: Identifier Clone Origin Categories Frequency Pie Chart

detected in our study were used consistently in representing the same entity. The "Non-Traveling" clones represent names that should potentially be refactored if developers reading them cannot tell from the surrounding context how to narrow down the meaning of the identifier name.

## 5.6 Semantic Relationships

The ambiguity present in Natural language due to semantic relationships between words such as hypernyms, hyponyms, synonyms, and homonyms makes the identifier naming process challenging (picking high quality terms to represent an entity) and hinders the comprehension of code when performing static code analysis. In our research study, we observed these semantic relationships being a potential cause for the introduction of identifier clones. Specifically, we noticed that parent-child relationships (hypernyms and hyponyms) present in the domain terminology of a system leads to imprecise terminology being used in the naming of identifiers. Which in turn leads to identifier clones using imprecise terminology. The following example is a clone introduced into SimianArmy due to this semantic relationship.

**Hypernym and Hyponym** Example#1. Identifier clone "resource" was cloned 80 times in SimianArmy repository and classified as "Imprecise" in terms of Conciseness and "Non-Domestic Traveling" in terms of Consistency. The function block containing this clone instance can be seen in Listing 5.5. The name "resource" is a hypernym of the following set of concrete aws resources: Instance, Elastic Block Storage Volume, Elastic Block Storage Snapshot, Auto Scaling Group, Launch Configuration, S3 Bucket, Security Group, Image, and Elastic Load Balancer. The semantic relationships between the concepts in the domain of the system, with "resource" being the parent concept of the set of concrete resources, translated into the design decisions of how the developers represented the aws resources in code. Developers created a class "AWSResource" that contains the instance field "resourceType" to implement this hypernym and hyponym relationship. During runtime, the class AWSResource can represent any concrete type of resource. This created an inconsistent naming pattern in how developers named identifiers storing aws resources. Sometimes, developers would use the name that belongs to the concrete resource type (i.e. "ami" standing for Amazon Machine Image as seen in Listing 5.6), while other times developers would use the name that belongs to the parent concept "resource". An improvement on this naming behavior, aligning to our definition of conciseness, would be to use the identifier name that belongs to that specific hyponym (i.e. "ami") if the code containing that variable only deals with a specific subtype. However, if due to polymorphism, a piece of code deals with any type of subtype during runtime, then the variable name should be kept to describing the hypernym entity (i.e. "resource").

```
    private Resource parseJsonElementToSnapshotResource(String region, JsonNode
        jsonNode) {
        long startTime = jsonNode.get("startTime").asLong();

        Resource resource = new
            AWSResource().withId(jsonNode.get("snapshotId").getTextValue())
                .withRegion(region)
                .withResourceType(AWSResourceType.EBS_SNAPSHOT)
                .withLaunchTime(new Date(startTime));
        JsonNode tags = jsonNode.get("tags");


        for (Iterator<JsonNode> it = tags.getElements(); it.hasNext();) {
            JsonNode tag = it.next();
            String key = tag.get("key").getTextValue();
            String value = tag.get("value").getTextValue();
            resource.setTag(key, value);
        }
        JsonNode description = jsonNode.get("description");
        ((AWSResource) resource)
            .setAWSResourceState(jsonNode.get("state").getTextValue());
        Collection<String> amis = snapshotToAMIs.get(resource.getId());
        resource.setOwnerEmail(getOwnerEmailForResource(resource));
        return resource;
    }
```

Listing 5.5: Hypernymy naming anti-pattern example. Identifier clone instance where hypernym term "resource" is used to represent the concrete resource "Elastic Block Storage Snapshot".

```java
    private void updateReferenceTimeByInstance(String region, List<Resource>
        batch, long since) {
        String batchUrl = getInstanceBatchUrl(region, batch, since);
        Map<String, Resource> idToResource = Maps.newHashMap();
        for (Resource resource : batch) {
            idToResource.put(resource.getId(), resource);
        }
        JsonNode batchResult = eddaClient.getJsonNodeFromUrl(batchUrl);

        for (Iterator<JsonNode> it = batchResult.getElements(); it.hasNext();) {
            JsonNode elem = it.next();
            JsonNode data = elem.get("data");
            String imageId = data.get("imageId").getTextValue();
            String instanceId = data.get("instanceId").getTextValue();
            JsonNode ltimeNode = elem.get("ltime");
            if (ltimeNode != null && !ltimeNode.isNull()) {
                long ltime = ltimeNode.asLong();
                Resource ami = idToResource.get(imageId);
                String lastRefTimeByInstance = ami.getAdditionalField(
                        AMI_FIELD_LAST_INSTANCE_REF_TIME);
                if (lastRefTimeByInstance == null ||
                    Long.parseLong(lastRefTimeByInstance) < ltime) {
                    LOGGER.info(String.format("The last time that the image %s was
                        referenced by instance %s is %d",
                            imageId, instanceId, ltime));
                    ami.setAdditionalField(AMI_FIELD_LAST_INSTANCE_REF_TIME,
                        String.valueOf(ltime));
                }
            }
        }
    }
```

Listing 5.6: Hypernymy inconsistent naming behavior example. Identifier clone instance where "ami" is used to represent the concrete resource "Amazon Machine Image" instead of the parent concept term "resource".

We also noticed that homonyms, words that are spelled the same but have multiple meanings, were a source of identifier clones classified as "Non-Traveling" (not used consistently). This is a naming anti-pattern as it means that developers may read the same word and misinterpret it for any one of its different meanings. These occurrences of identifier clones were tied to our "Identifier Origin" categories as we observed that some clones had multiple meanings depending on what resource, or origin, one must refer to in order to interpret its correct meaning. The following example is a clone that follows this naming anti-pattern due to the homonym semantic relationship.

**Homonym** Example#1. Identifier clone "node" was detected in SimianArmy and classified as "Non-Traveling" (not used consistently). The meaning of the word "node" in two clone instances represents an aws instance node and was used in the context of establishing an ssh connection to these nodes. The origin for this meaning comes from the Project Requirements or Project Domain. The meaning of the word "node" in the other 3 clone instances represent a json node element in a json object fetched from an external data source. The origin for this meaning comes from Developer terminology. We can see that the word "node" is overloaded with two different meanings coming from different origins that interpret their meaning differently.

```
public static Map<String, String> getAllApplicationOwnerEmails(EddaClient
    eddaClient) {
    String region = "us-east-1";
    LOGGER.info(String.format("Getting all application names and emails in
        region %s.", region));

    String url = eddaClient.getBaseUrl(region) +
        "/netflix/applications/;_expand:(name,email)";
    JsonNode jsonNode = eddaClient.getJsonNodeFromUrl(url);

    Iterator<JsonNode> it = jsonNode.getElements();
    Map<String, String> appToOwner = new HashMap<String, String>();
    while (it.hasNext()) {
        JsonNode node = it.next();
        String appName = node.get("name").getTextValue().toLowerCase();
        String owner = node.get("email").getTextValue();
        if (appName != null && owner != null) {
            appToOwner.put(appName, owner);
        }
    }
    return appToOwner;
}
```

Listing 5.7: Homonym naming anti-pattern example. Identifier clone instance where "node" is used to represent a json node element.

```
@Override
public SshClient connectSsh(String instanceId, LoginCredentials credentials) {
    ComputeService computeService = getJcloudsComputeService();

    String jcloudsId = getJcloudsId(instanceId);
    NodeMetadata node = getJcloudsNode(computeService, jcloudsId);

    node = NodeMetadataBuilder.fromNodeMetadata(node)
        .credentials(credentials).build();

    Utils utils = computeService.getContext().utils();
    SshClient ssh = utils.sshForNode().apply(node);

    ssh.connect();

    return ssh;
}
```

Listing 5.8: Homonym naming anti-pattern example. Identifier clone instance where "node" is used to represent an aws instance being connected to through ssh.

1. Probability that a clone will be non-traveling (not used consistently) given that it is generic 2. Linguistic patterns introducing clones (homonyms, hypernyms & hyponyms)

What are some interesting relationships observed between the sets of categories?

For the small set of identifier clones that were classified as "Generic Identifiers", all of these clones were also classified as having the origin of "Natural Language" (developers must reference an English dictionary to determine the correct meaning of the terms in the identifiers). It makes sense that we did not observe identifier clones classified as both Generic and having an origin of Developer or Project terminology. This is because Developer and Project terminology holds more contextual information that would increase the conciseness of an identifier. Therefore, it is highly likely that the majority of Generic identifiers we encounter will have a generic natural language term such as "value", or "data".

# Chapter 6

# Threats to Validity

In this section, we discuss threats to the validity of our results. These threats are broken into three separate categories [95].

## 6.1 Internal Validity

Internal validity is concerned with how confident we are that our final clone categories have a true cause-and-effect relationship with the introduction of identifier clones into a system, and that identifier clones are not the cause of other external factors not measured or theorized in our study.

This is a concern in the resulting taxonomy we propose since the background of the researcher making the grounded observations, and establishing relationships between the data observed, impacts what categories are conceptualized and how the final taxonomy is broken down. Our weekly meetings between the two researchers involved in conceptualization of clone categories counters this risk. However, there is a risk that limitations in our experience or background prevented us from establishing relationships seen during static code analysis.

Another concern is the limited scope of the source code analyzed for codings. Static code analysis was only performed on the function blocks where identifier clone instances were declared either as a function parameter or a local variable. We did not expand this scope to include recording the characteristics of the classes or subsystems containing the clones. Therefore, there could be additional sources of information that provide additional insight as to why identifier clones are introduced into a system.

Further, when analyzing the identifier clones present in the "SimianArmy" repository, we did not include all detected clones. Instead, we only analyzed a sample with 95% Confidence Level. Therefore, there could be some important characteristics of identifier names that we did not observe in our sample.

## 6.2 Construct Validity

Construct validity is concerned with the validity of the measurements we used to derive our results. Specifically for our research, whether our measurements of clone conciseness, consistency, identifier behavior stereotypes, and origin are well-designed to classify identifier clones into our proposed categories. Our measurements for deriving conciseness, consistency, and identifier behavior categories were based on empirical data gathered from static code analysis. For example, an identifier that is used to modify a fetch query to an external data source is classified as "External Accessor Modifier" given that we observed that this was the usage of the identifier in the source code. This is a concern as our Identifier Behavior Stereotypes may not be fully complete given that they are the result of analyzing the usage of only 1,432 identifiers across only two systems. Analyzing further systems in different domains performing additional unseen behavior would expand our proposed categories.

Our measurement for determining whether clones are consistently used across a system is a concern due to not having access to the original developer that introduced that clone into the system. We determined the meaning and usage of identifier clones through static code analysis, which we used to determine the consistency of clones. However, additional information such as the original intent of the developer would provide supporting evidence whether the clone is indeed used consistently and represents the same entity in each declaration sharing a name.

Our measurement for determining the clone origin categories (source that a developer must refer to understand the correct meaning of an identifier) is based on analyzing the terms used in the identifier name and determining whether those terms are Natural Language, Project, or Developer terms based on how the identifier was used in the code. However, there could be terms that are not classified correctly given that natural language contains ambiguous semantic relationships such as homonyms, where the same term can have different meanings depending on context and source for the term (i.e. English dictionary, or Project Requirements). Again, having access to the original developer that introduced the identifier clone would prevent these classification errors as they could confirm the true meaning of the identifier and what sources the terms come from.

## 6.3 External Validity

External validity is concerned with whether our final clone categories are able to generalize to unseen data (identifier clones in other systems). Given that our study was limited to analyzing identifier clones present in Java systems, we are at risk of not generalizing to clones present in systems developed in other programming languages. This concern is elevated for generalizing our outcomes to identifier clones present in programming languages that do not follow an Object-Oriented Programming model like Java, and instead follow the procedural programming model. This is a concern because the identifier naming process can be affected by the features of a programming language. For example, polymorphism in java will prompt developers to represent domain concepts in code using a hierarchical structure of classes, which simulate hypernym and hyponym semantic relationships that were seen to be a potential cause for the introduction of identifier clones.

Our results are also at risk of not generalizing to other systems varying in domain, development team, and size. Given that we only analyzed two systems, which in total had 9 main contributors, our research is only observing the identifier naming performed by a small number of developers. This is a risk as other studies have found that naming is not consistent between different developers [6]. For our categories to generalize to a more representative group of naming practices, we need to increase the number of identifier names we analyze that are the outcome of a larger sample of developers.

# Chapter 7

# Conclusion & Future Work

The objective of this work is to help support the understanding of identifier naming phenomenon that impact the quality of identifier names and can help in detecting naming anti-patterns. To do so, we have created a taxonomy of identifier clones that can characterize clones in terms of conciseness, consistency, origin, and behavior in order to give insight as to why identifier clones are introduced into systems and whether they are a source of naming anti-patterns. We conducted an empirical study, following the Grounded Theory research methodology, on identifier clones detected in software engineered open-source systems. We derived four sets of categories ("Conciseness", "Traveling Clones", "Identifier Origin", and "Identifier Behavior Stereotypes") that are non-mutually exclusive, meaning each identifier clone is classified into a category in each of these sets. Our main findings indicate the distribution of clones based off of our classification, providing a better picture of what identifier clones look like and what properties they have. We also make connections between some types of identifier clones and naming anti-patterns hindering program comprehension.

We plan on extending this study by analyzing identifier clones present in additional software systems. We also plan on: (1) Working towards automating the classification of identifier clones into our proposed categories, (2) exploring cross-system clones versus clones that are only local to specific systems, and (3) exploring identifier clones that are not exact clones but that share some terms within their name. Potentially finding similar head noun indicating they share a common parent concept (hypernymy).

# Bibliography


[1] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

[2] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE, 2015.

[3] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[4] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

[5] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*, pages 271–278. IEEE, 2002.

[6] Dror Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. How developers choose names. *IEEE Transactions on Software Engineering*, 2020.

[7] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*, pages 217–227. IEEE, 2017.

[8] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE International*

*Conference on Program Comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

[9] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 31–3109. IEEE, 2018.

[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[11] Christian D Newman, Reem S AlSuhaibani, Michael J Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740, 2020.

[12] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[13] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33, 2018.

[14] Christian D Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, 2021.

[15] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[16] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, page e2395, 2021.

[17] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[18] Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian Newman, and Mohamed Wiem Mkaouer. Satdbailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022.

[19] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):1–43, 2022.

[20] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. On the documentation of refactoring types. *Automated Software Engineering*, 29(1):1–40, 2022.

[21] Eman Abdullah Alomar, Tianjia Wang, Vaibhavi Raut, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: an empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2022.

[22] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D Newman. Using grammar patterns to interpret test method name evolution. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 335–346. IEEE, 2021.

[23] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Mining and managing big data refactoring for design improvement: Are we there yet? *Knowledge Management in the Development of Data-Intensive Systems*, pages 127–140, 2021.

[24] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif

Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[25] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 347–357. IEEE, 2019.

[26] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[27] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. *arXiv preprint arXiv:1906.03060*, 2019.

[28] Anthony Peruma, Mohamed Wiem Mkaouer, Michael John Decker, and Christian Donald Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–85. IEEE, 2019.

[29] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 34–41, 2018.

[30] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *CASCON*, pages 60–71, 2018.

[31] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems*, pages 295–320. Auerbach Publications, 2019.

[32] Montassar Ben Messaoud, Ilyes Jenhani, Nermine Ben Jemaa, and Mohamed Wiem Mkaouer. A multi-label active learning approach for mobile

app user review classification. In *International Conference on Knowledge Science, Engineering and Management*, pages 805–816. Springer, 2019.

[33] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. Migrationminer: An automated detection tool of third-party java library migration at the method level. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 414–417. IEEE, 2019.

[34] Deema Alshoaibi, Kevin Hannigan, Hiten Gupta, and Mohamed Wiem Mkaouer. Price: Detection of performance regression introducing code changes using static and dynamic metrics. In *International Symposium on Search Based Software Engineering*, pages 75–88. Springer, 2019.

[35] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[36] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, pages 1–17, 2021.

[37] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. IEEE, 2019.

[38] Alex Bogart, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Increasing the trust in refactoring through visualization. In *2020 IEEE/ACM 4th International Workshop on Refactoring (IWoR)*, 2020.

[39] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[40] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In

*2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[41] Eman Abdullah AlOmar, Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 342–349, 2020.

[42] Eman Abdullah AlOmar, Philip T Rodriguez, Jordan Bowman, Tianjia Wang, Benjamin Adepoju, Kevin Lopez, Christian Newman, Ali Ouni, and Mohamed Wiem Mkaouer. How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer, 2020.

[43] Eman Abdullah AlOmar, Tianjia Wang, Raut Vaibhavi, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. Refactoring for reuse: An empirical study. *Innovations in Systems and Software Engineering*, pages 1–31, 2021.

[44] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 350–357, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[47] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages using multi-label active learning. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1760–1767, 2019.

[48] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[49] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 331–336, 2014.

[50] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1263–1270, 2014.

[51] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering*, 21(6):2503–2545, 2016.

[52] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering*, 22(2):894–927, 2017.

[53] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, and Ali Ouni. Recommending relevant classes for bug reports using multi-objective search. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 286–295. IEEE, 2016.

[54] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 193–202, 2019.

[55] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29(1):1–61, 2022.

[56] Wajdi Aljedaani, Mona Aljedaani, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Stephanie Ludi, and Yousef Bani Khalaf. I cannot see you—the perspectives of deaf students to online learning during covid-19 pandemic: Saudi arabia case study. *Education Sciences*, 11(11):712, 2021.

[57] Islem Saidani, Ali Ouni, and Wiem Mkaouer. Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, 2021.

[58] Marwa Daaji, Ali Ouni, Mohamed Mohsen Gammoudi, Salah Bouktif, and Mohamed Wiem Mkaouer. Multi-criteria web services selection: Balancing the quality of design and quality of service. *ACM Transactions on Internet Technology (TOIT)*, 22(1):1–31, 2021.

[59] Nuri Almarimi, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. csdetector: an open source tool for community smells detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1560–1564, 2021.

[60] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Bf-detector: an automated tool for ci build failure detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1530–1534, 2021.

[61] Oumayma Hamdi, Ali Ouni, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. A longitudinal study of the impact of refactoring in android applications. *Information and Software Technology*, 140:106699, 2021.

[62] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, Mel Ó Cinnéide, and Mohamed Wiem Mkaouer. An empirical study on the impact of refactoring on quality metrics in android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39. IEEE, 2021.

[63] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 138:106618, 2021.

[64] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology*, 135:106566, 2021.

[65] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. On the classification of bug reports to improve bug localization. *Soft Computing*, 25(11):7307–7323, 2021.

[66] Makram Soui, Mabrouka Chouchane, Narjes Bessghaier, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the impact of aesthetic defects on the maintainability of mobile graphical user interfaces: An empirical study. *Information Systems Frontiers*, pages 1–18, 2021.

[67] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[68] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. Anti-patterns in modern code review: Symptoms and prevalence. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 531–535. IEEE, 2021.

[69] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, 25(7):5619–5632, 2021.

[70] Hussein Alrubaye, Deema Alshoaibi, Eman Alomar, Mohamed Wiem Mkaouer, and Ali Ouni. How does library migration impact software quality and comprehension? an empirical study. In *International Conference on Software and Software Reuse*, pages 245–260. Springer, 2020.

[71] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing*, 100:106908, 2021.

[72] Moataz Chouchen, Ali Ouni, and Mohamed Wiem Mkaouer. Androlib: Third-party software library recommendation for android applications. In *International Conference on Software and Software Reuse*, pages 208–225. Springer, 2020.

[73] Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. On the detection of community smells using genetic programming-based ensemble classifier chain. In *Proceedings of the 15th International Conference on Global Software Engineering*, pages 43–54, 2020.

[74] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821, 2021.

[75] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. Web service api anti-patterns detection as a multi-label learning problem. In *International Conference on Web Services*, pages 114–132. Springer, 2020.

[76] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. Learning to rank developers for bug report assignment. *Applied Soft Computing*, 95:106667, 2020.

[77] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[78] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[79] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. tsdetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[80] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. On the prediction of continuous integration build failures using search-based software engineering. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 313–314, 2020.

[81] Nuri Almarimi, Ali Ouni, and Mohamed Wiem Mkaouer. Learning to detect community smells in open source software projects. *Knowledge-Based Systems*, 204:106201, 2020.

[82] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In *International Conference on Service-Oriented Computing*, pages 58–63. Springer, Cham, 2019.

[83] Nuri Almarimi, Ali Ouni, Salah Bouktif, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Mohamed Aymen Saied. Web service api recommendation for automated mashup creation using multi-objective evolutionary search. *Applied Soft Computing*, 85:105830, 2019.

[84] Makram Soui, Mabrouka Chouchane, Mohamed Wiem Mkaouer, Marouane Kessentini, and Khaled Ghedira. Assessing the quality of mobile graphical user interfaces using multi-objective optimization. *Soft Computing*, 24(10):7685–7714, 2020.

[85] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. Learning to rank faulty source files for dependent bug reports. In *Big Data: Learning, Analytics, and Applications*, volume 10989, page 109890B. International Society for Optics and Photonics, 2019.

[86] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ocinneide, Ali Ouni, and Yuanfang Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961, 2018.

[87] Makram Soui, Mabrouka Chouchane, Ines Gasmi, and Mohamed Wiem Mkaouer. Plain: Plugin for predicting the usability of mobile user interface. In *VISIGRAPP (1: GRAPP)*, pages 127–136, 2017.

[88] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference on the Applications of Evolutionary Computation*, pages 20–34. Springer, Cham, 2017.

[89] Mohamed Wiem Mkaouer. Interactive code smells detection: An initial investigation. In *International Symposium on Search Based Software Engineering*, pages 281–287. Springer, Cham, 2016.

[90] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. A robust multi-objective approach for software refactoring under uncertainty. In *International Symposium on Search Based Software Engineering*, pages 168–183. Springer, Cham, 2014.

[91] Mohamed Wiem Mkaouer and Marouane Kessentini. Model transformation using multiobjective optimization. In *Advances in Computers*, volume 92, pages 161–202. Elsevier, 2014.

[92] Mohamed W Mkaouer, Marouane Kessentini, Slim Bechikh, and Daniel R Tauritz. Preference-based multi-objective software modelling. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 61–66. IEEE, 2013.

[93] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, pages 120–131, 2016.

[94] Michael J Decker, Christian D Newman, Natalia Dragan, Michael L Collard, Jonathan I Maletic, and Nicholas A Kraft. Which method-stereotype changes are indicators of code smells? In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 82–91. IEEE, 2018.

[95] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering.* Springer Science & Business Media, 2012.