UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Remo de Oliveira Gresta

Naming Analysis: Exploring Practices in Object-Oriented Programming

São João del-Rei

2024

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Remo de Oliveira Gresta

Naming Analysis: Exploring Practices in Object-Oriented Programming

Dissertação apresentada como requisito para obtenção do título de mestre em Ciências no Curso de Mestrado do Programa de Pós Graduação em Ciência da Computação da UFSJ.

Supervisor: Elder José Reioli Cirilo

Universidade Federal de São João del-Rei – UFSJ

Mestrado em Ciência da Computação

São João del-Rei

2024

Remo de Oliveira Gresta

Naming Analysis: Exploring Practices in Object-Oriented Programming

Dissertação apresentada como requisito para obtenção do título de mestre em Ciências no Curso de Mestrado do Programa de Pós Graduação em Ciência da Computação da UFSJ.

Elder José Reioli Cirilo
Federal University of São João del-Rei

Vinícius Humberto Serapilha Durelli
Federal University of São João del-Rei

Bruno Barbieri de Pontes Cafeo
State University of Campinas

São João del-Rei
2024

Acknowledgements

First of all, I'd like to thank all my family, my mother Simone, my grandparents Hélio and Araci, my partner Thauana, my father Romolo, my friends, and all the people that support me all these years and made me who I am. Finally, I'd like to also thank all members of the proletarian class, the worker class, those who actually transform the world, we will win!

"Workers of the world, unite!"
- Karl Marx

Abstract

Currently, research indicates that comprehending code takes up far more developer time than writing code. Given that most modern programming languages place little to no limitations on identifier names, and so developers are allowed to choose identifier names at their own discretion, one key aspect of code comprehension is the naming of identifiers. Research in naming identifiers shows that informative names are crucial to improving the readability and maintainability of programs: essentially, intention-revealing names make code easier to understand and act as a basic form of documentation. Poorly named identifiers tend to hurt the comprehensibility and maintainability of software systems. However, most computer science curricula emphasize programming concepts and language syntax over naming guidelines and conventions. Consequently, programmers lack knowledge about naming practices. This study is an extension of our previous study on naming practices. Previously, we set out to explore naming practices of Java programmers. To this end, we analyzed 1,421,607 identifier names (i.e., attributes, parameters, and variables names) from 40 open-source Java projects and categorized these names into eight naming practices. As a follow-up study to further investigate naming practices, we examined 40 open-source C++ projects and categorized 1,181,774 identifier names according to the previously mentioned eight naming practices. We examined the occurrence and prevalence of these categories across C++ and Java projects and our results also highlight in which contexts identifiers following each naming practice tend to appear more regularly. Finally, we also conducted an online survey questionnaire with 52 software developers to gain insight from the industry. All in all, we believe the results based on the analysis of 2,603,381 identifier names can be helpful to enhance programmers' awareness and contribute to improving educational materials and code review methods.

Keywords: Naming Identifiers, Program Comprehension, Mining Software Repositories, Static Code Analysis

List of Figures

List of Tables

Contents

1 Introduction

Reading and comprehending source code plays a vital role in software development (ALLAMANIS, 2014), especially when documentation is scarce or not available. Indeed, several researchers have shown the value of comprehensible source code and their relevance in software development tasks (BUTLER, 2010) (AVIDAN; FEITELSON, 2017) (HOFMEISTER, 2017). Evidence suggests that choosing proper names for identifiers can positively impact code comprehension (LAWRIE, 2007) (FAKHOURY, 2018). Therefore, an important aspect of software comprehension is to understand the underlying concepts embodied in the code by means of decoding identifier names (AVIDAN; FEITELSON, 2017). Although giving meaningful names to identifiers is a widely accepted best practice, coming up with proper names is challenging (DEISSENBOECK; PIZKA, 2006). As stated by Host and Ostvald(HOST; OSTVOLD, 2007), even though naming is part of daily life for programmers, it entails a great deal of time and thought: names should convey to others the purpose of the code (MARTIN, 2008) and reflect the meaning of domain concepts (MARCUS, 2004). Meaningful identifier names are key to bridging the gap between intention and implementation (WAINAKH, 2021). Therefore, given that poorly chosen identifier names might hinder source code comprehension (SCHANKIN, 2018), using meaningful identifier names is a recommended practice present in several coding style guides and conventions.

According to the Java language naming conventions, names should be "short yet meaningful" [1]. In a similar fashion, the Google C++ style guide states that names should be "as descriptive as possible" [2]. Martin argues that programmers should choose intention-revealing names as a way to avoid disinformation (MARTIN, 2008). He also advocates that names have to contain meaningful distinctions and be descriptive (not abbreviated). The GNU Coding Standards posit that programmers should not "choose terse names – instead, [they should] look for names that give useful information about the meaning of the variable". Although programming communities and internationally renowned experts have proposed best practices related to naming identifiers, little is known about the extent to which programmers follow these naming practices (ARNAOUDOVA, 2016).

We argue that without proper guidance, programmers are more likely to adopt sub-optimal naming practices, such as using number series or noise words. For instance, poor naming practices might create the misconception that names like `Person person1` and `Person person2` are intuitive and understandable. Such careless naming can hinder not only code comprehension but also overall team communication. Indeed, the importance

[1] oracle.com/java/technologies/javase/codeconventions-namingconventions.html
[2] google.github.io/styleguide/cppguide.html

of meaningful identifier names has been established in several studies. For example, research has shown that full-word identifiers lead to better comprehension than single-letter identifiers (HOFMEISTER, 2017). However, a significant portion of source code vocabulary consists of acronyms, abbreviations, or concatenations of terms that are not easily identifiable and do not follow any naming convention (DEISSENBOECK; PIZKA, 2006). Another study found statistically significant associations between poor-quality names and code quality issues reported by static analysis tools like FindBugs (BUTLER, 2010). As expected, names chosen by one developer may not convey the intended meaning in a collaborative context, impairing software comprehension. In recent works, experiments were conducted on identifier naming and its impact on software maintenance in collaborative environments. The first study focused on the semantic similarity in identifier naming (GRESTA; CIRILO, 2020). It was observed that source code generally maintains an acceptable level of contextual similarity, with developers avoiding names outside the default dictionary (e.g., domain-specific terms). Files with more changes and contributions from multiple developers tend to have better contextual similarity. In a subsequent study, the phonetic similarity of names and the complexity of hard-to-pronounce English words were analyzed (GRESTA; CIRILO, 2021). It was found that many analyzed names contain hard-to-pronounce words, leading to a significant overall word complexity score in projects. Therefore, we argue that it is crucial for software engineering researchers to learn how to support programmers by understanding how naming practices are used "in the wild" and, through this better understanding, defining naming guidelines for educational materials (CHARITSIS, 2021) and code review (NYAMAWE, 2021).

To investigate how C++ and Java programmers name attributes, parameters, and variables, we carried out an empirical study in which we analyzed 1,421,607 identifier names from 40 open-source Java projects and 1,181,774 identifier names from 40 open-source C++ projects. In the first part of the study, we used mined repositories to determine how often eight categories of naming practices are within and across these projects. We also looked at how prevalent these naming practices are in certain code contexts (i.e., ATTRIBUTE, PARAMETER, METHOD, FOR, WHILE, IF, and SWITCH). Moreover, to understand the industry practices, we conducted an online survey questionnaire to gain insight from software programmers. Throughout a survey, we gathered quantitative data on programmers' perceptions about the use and occurrence of the investigated naming practices. The online survey questionnaire ran from November 2021 to January 2022 and had 52 responses.

In this work, we also developed a static analysis tool that evaluates identifier names, categorizing them based on naming practices identified in our empirical study. Our goal with this tool is to increase awareness of poor naming conventions among developers, promoting better practices and improving code quality. The tool scrutinizes the morphological aspects of words used in identifier names, assessing the potential impact of

naming patterns on the overall quality of the code. Recognizing the crucial role of identifier names in software development (BUTLER, 2010), and the significant improvements possible through continuous testing and validation, we have implemented our tool as a plugin to a CI/CD pipeline.

As a result of our study, we can make the following contributions:

- Developers tend to preserve an acceptable level of contextual similarity among names in the source code
- Developers tend to use existing words to name identifiers, that is, they usually avoid the use of out-of-the-dictionary (e.g., domain) words
- Our results show that the naming practice categories (Kings, Median, Ditto, Diminutive, Cognome, Shorten, Index, and Famed) appear in all 80 open-source projects and are prevalent in practice;
- We identified the most common names across projects. The Top-3 recurrent names are value; result; and name. Many single-letter names are also commonly used in projects (e.g., i, e, s, c). We also observed that the majority of common names are associated with integer or string values;
- We perceived that programmers' naming practices are context-specific. Single-letter names (Index and Shorten) seem to be more present in conditional or loop statements (IF, FOR, WHILE). In contrast, identifiers with the same name as their Types tend to appear in large-scope contexts (e.g., ATTRIBUTE);
- We noted that, in general, the project's characteristics might not impact the prevalence of one particular naming category practice;
- We observed that Diminutive is the most adopted naming category practice by survey respondents and Median is the least one. This result seems to align well with our observation about the prevalence of naming practices in 80 open-source object-oriented programs.
- The development of a tool used to detect the presence of eight different naming categories that could be harmful to the code's comprehension.

2 Background

This chapter presents some background about identifier naming, names, and Continuous Integration and Continuous Deployment process. We introduce this section by presenting an overview of the role of names in software development and their importance in the field.

2.1 Identifier Names

2.1.1 Naming

According to Deissenboeck et al. (DEISSENBOECK; PIZKA, 2006), approximately two-thirds of any regular source code is composed of identifier names. Names identify classes, attributes, methods, variables, and parameters (LAWRIE, 2006), but they are also, alongside comments, the main sources of domain information. Originally designed to represent values in memory (TOFTE; TALPIN, 1997), identifier names have now become the primary source of information in software development (LAWRIE, 2006)(DEISSENBOECK; PIZKA, 2006). Programmers rely on existing names in their code comprehension process (TAKANG, 1996). An identifier's name can be a fully spelled word, an abbreviation of a word, or a combination of two or more words. Names might also involve words that do not actually exist or even be single alphabetical characters. In general, a fully spelled word is more descriptive than a single character. Indeed, the proper use of names has been recognized as a major issue in software development. Lawrie et al. (LAWRIE, 2006), have shown that the use of full-word names for identifiers assists developers in maintainability tasks. In their study, developers were able to better understand code snippets in contexts where full words were chosen as names, instead of single-letter variants. Naturally, choosing low-quality or unrelated names results in source code that is more difficult to maintain and can be associated with bugs (LI, 2018)(KAWAMOTO; MIZUNO, 2012)(BUTLER, 2010).

Indeed, high-quality names have a significant influence on the comprehension of source code (AVIDAN; FEITELSON, 2017). Arnaoudova et al. (2016) have acknowledged the critical role that the source code lexicon plays in the psychological complexity of software systems and coined the contradictory expression "Linguistic Antipatterns" (LAs) to denote poor practices in the naming, documentation, and choice of identifiers that might hinder program understanding (ARNAOUDOVA, 2016). They argue that poor practices might lead programmers to make wrong assumptions and waste time understanding source code (ARNAOUDOVA, 2016). Deissenboeck and Pizka (2006) characterized a name as be-

ing a fully spelled word or even an abbreviation (DEISSENBOECK; PIZKA, 2006). Names can also be composed of two or more words, might include words that do not exist, or even be single alphabetical characters. However, the proper use of words in names is a significant issue in software development (FEITELSON, 2020). In Martin's book (MARTIN, 2008), Tim Ottinger drew a series of simple rules to guide programmers on naming identifiers. According to Ottinger, programmers have to focus on creating intention-revealing names (the name by itself should be capable of informing what it does). They also have to avoid using non-informative words (e.g., words with multiple meanings, words with little differentiation between themselves or number series). Ottinger also advocates that names should be pronounceable and searchable. For instance, it is impractical to discuss any source code composed of words that programmers cannot pronounce in a code review session. Coding style guides and conventions also aim to address the naming identifiers' challenges (SANTOS; GEROSA, 2018). However, they are usually hard to enforce rules, as others discussed in Martin's book (Clean Code) (MARTIN, 2008). Caprile and Tonella (2000) proposed an approach for improving the meaningfulness of identifier names (CAPRILE; TONELLA, 1999). The approach entails the following steps: (i) extracting identifier names; (ii) normalizing identifier names; and (iii) applying the changes to the source code. The proposed rules for creating meaningful names aim to guarantee that each word composing a name must belong to a dictionary of standard words and be compliant with existing grammar. Deissenboeck and Pizka (2006) proposed a set of precise rules for constructing concise and consistent names (DEISSENBOECK; PIZKA, 2006). In the interest of preserving consistency, the authors advocate that a single name must represent only one concept. The rules, therefore, ensure that one concept will not be taken into consideration in multiple identifier names. In order to preserve conciseness, the rules ensure that names chosen by programmers stand for the concepts they are indeed trying to convey. More recently, Feitelson et al. (2020) suggested a three step method to help programmers to systematically come up with meaningful names. The model encompasses the following steps: (i) selecting the concepts to include in the name; (ii) choosing the words to represent each concept; and (iii) creating a name from these words (FEITELSON, 2020). The authors demonstrated that programmers could use the model to guide choosing names that are superior (in terms of meaningfulness) over randomly chosen names.

2.1.2 Names in Software Quality

Numerous studies have investigated the impact of naming on code comprehension and programmer efficiency. Avidan and Feitelson conducted an experiment in 2017 with ten programmers to understand the influence of identifier names on program comprehension (AVIDAN; FEITELSON, 2017). They found that identifiers with fully spelled words were perceived as more understandable compared to their single-letter counterparts. Simi-

larly, Hofmeister et al. concluded that abbreviations and single-letter names diminish code comprehension and could be indicative of low-quality code, aligning with observations by Butler et al. (2010) and Kawamoto and Mizuno (HOFMEISTER, 2017)(KAWAMOTO; MIZUNO, 2012)(BUTLER, 2010). Butler et al. demonstrated in 2010 that source code containing poor quality identifier names were associated with FindBugs warnings (BUTLER, 2010). Kawamoto and Mizuno noted that concise identifier names significantly affect fault-proneness in NetBeans (KAWAMOTO; MIZUNO, 2012). Takang et al. conducted a survey in 1996 with 89 computer science students, deduced that combining identifier names with comments in code marginally improves comprehension (TAKANG, 1996). Therefore, enhancing identifier names appears to be more beneficial than adding comments. Lawrie et al. observed that spending more time on selecting meaningful identifier names can reduce the workload during software maintenance (LAWRIE, 2007). Low-quality names can detrimentally impact code by causing confusion and misinformation. The study by Lawrie et al. (2007a) also found that the quality of identifier names improves over time and is associated with the software license. Modern software systems tend to contain higher-quality names, with proprietary software featuring more abbreviations than open-source projects. Furthermore, a study examining the semantic nature of identifier names in four large-scale open-source projects revealed that the number of commits and contributors tends to influence the quality of names. Projects with a high number of commits and contributors usually have more identifier names with a substantial text corpus of existing words (GRESTA; CIRILO, 2020).

2.2 Continuous Delivery and Integration

Continuous Integration and Continuous Delivery (CI/CD) are a set of practices in DevOps designed to optimize the quality process off releasing software. That includes software testing, frequency of commits per day, version control etc (SKA; SYED, 2019). These process range from developing stage, testing, production and monitoring. CI/CD are a part of the Continuous software engineering, that aims to develop, deploy and get quick feedback from software and customer in a fast cycle (SHAHIN, 2017). Alongside Continuous Integration and Delivery there is also Continuous Deployment, that aims to apply automatically deployment of the application into production.

One of the goals that CI intend to achieve is to have shorter release cycles, improving code quality and team productivity (SHAHIN, 2017). In CI, team members are encouraged to integrate and merge their work in short periods of time, including software testing and automate software building (SKA; SYED, 2019). The practice of putting together automate software build and test with faster integration's are a manner to discover in earlier stages possible bugs and code smells. Another advantage in performing continuous integration is to make it easier presenting and validating new releases of the

software. Continuous integration focus on committing to the code base more frequently, for example once commit a day in company of automated build and tests, such as running all unit tests and approving the commit only with a certain percentage of the code base was checked, and all test were passed. If all tests were passed, and the developer's team also implement Continuous Delivery, a set of acceptance tests are applied to the code, in order to check whether the code have any regression in any feature (RANGNAU, 2020). This behavior of often committing and testing make it easier to find bugs, improve code quality, reduced time for backtracking old code, and also reduce integration risks between coworkers (SHAHIN, 2017). With mode regularity in commits, the risk of having merge problems is lower, since that fewer lines of code are getting merged into the code base.

Inside the process of Continuous Delivery, a developer can integrate several static analysis tools to validate different aspects of the source code. Static analysis tools are widely used in the development of software and can, for example, be used to reduce bugs from security, memory, data typing to check coding styles and guidelines, reveal code smells, etc (LOURIDAS, 2006). A static source code analyzer tool analyzes the source code without actually executing it, they use data flow analysis, control flow analysis, interface analysis, and path analysis of source code, and are used to improve software quality by detecting potential defects and problematic code constructs in early development process (PRÄHOFER, 2012). A study that inspected the use of different static analysis tools found that those tools are an effective way of detecting critical defects, are a relatively affordable fault detection technique, and that a large percentage of errors made by developers detected by those tools had the potential to cause security vulnerabilities (ZHENG, 2006).

As noted above, static analysis tools have the ability to discover bugs, security issues, and inconsistencies in coding guidelines by analyzing the source code, without running the program (LOURIDAS, 2006). Because of that behavior, static analysis tools can integrate well in the first steps of a CI/CD pipeline. Using these tools within a pipeline ensures that they are going to be used automatically, preventing potential hazards in the program. Used in the early steps of a CI/CD pipeline, alongside automated tests, is a way to detect issues and correct them before the creation of pull requests and the need for code review. The next sections will demonstrate the development of a static analysis tool that focuses solely on inspecting identifier names chosen by developers. Regarding the importance of checking the source code for possible bugs and potential comprehension troubles before the process of building and deploying, our tool can be used to find potential naming hazards that can lead to difficulty in comprehending source code.

Through the consistence application of Continuous Integration, developers can achieve Continuous Delivery, in which software can be released to production at any time (FOWLER; FOEMMEL, 2006). With the aid of CI, like making small changes regularly,

with testing and quality checks, and alongside deployment automation to allowing the application to be always in a production-ready state (SHAHIN, 2017). Continuous Delivery is performed right after a successfully Continuous Integration process, and followed by a preparation for building and deploying to a desired stage. This process enable the developer to test the software with real-life scenarios, for example with integration tests, UI tests, depending on the desired amount of tests in each production stage (SINGH, 2019).

Continuous Delivery can be achieved by constantly applying CI practices and maintaining a real-time infrastructure that monitors, give alerts, that can give fast feedback, often through a CI/CD Pipeline. This Pipeline is used to automate the process of building and testing the application, and is separated into different stages. Each stage is responsible for a part of the process, such as build, test, quality check, merge and deployment (FOWLER; FOEMMEL, 2006). The main difference between Continuous Delivery and Continuous Deployment is that in the second one every change that is applied to the pipeline gets into production, and the first the merge into production can happen, the code is at a state of production-ready, but not necessarily is merged into production, due to integration strategies. Applying correctly the principles of CD in a software project can create benefits to the development, such as a reduced deployment risk, believable progress and quick user feedback (FOWLER; FOEMMEL, 2006). These befits leverages the CDs smaller changes enabling more control over the process of validating changes, without wasting time developing a whole new feature with no quick feedback.

3 Exploring Naming Practices in Object-Oriented Programming

We conducted an empirical study to characterize how C++ and Java programmers name attributes, parameters, and variables. Specifically, we analyzed 1,421,607 identifier names (i.e., attributes, parameters, and variables names) from 40 Java projects and categorized these names into eight naming practice categories. Afterwards, we expanded our analysis by selecting a sample of 40 C++ projects. Upon analyzing this sample, we found 1,181,774 identifier names, which we then categorized according to the aforementioned eight naming practice categories. We used the results of categorizing identifier names from these two samples to provide answers to the research questions.

3.1 Goal and Research Questions

We set out to probe into how common eight naming practices are "in the wild" (i.e., in real world software systems) – see Section 3.2. Each category was created by us, based on coding conventions and guidelines, and each one of them has the potential to produce hazards in software development. More specifically, our goal is to contribute towards a better understanding of their prevalence in attributes, parameters, and variables naming in Java. We believe a more insightful interpretation of the results of our study can be obtained from the standpoint of a researcher interested in helping programmers by defining naming guidelines for educational material and code review and aiding the development of tools that can contribute to identifying potential naming hazards. Our main goal is to provide answers to the following research questions (RQs):

- **RQ_1: *How prevalent are the eight naming practices categories?***

  We set out to investigate whether identifier names in open-source projects can be categorized according to eight naming practices categories and how common these naming practices are across C++ and Java projects;

- **RQ_2: *Are there context-specific naming practices categories?***

  We set out to examine if specific naming practice categories tend to occur more often in certain contexts (e.g., ATTRIBUTE, PARAMETER, METHOD, IF, FOR, WHILE, SWITCH);

- **RQ$_3$: *Do the naming practice categories carry over across different C++ and Java projects?***

  We attempt to explore the prevalence of the categories spanning multiple C++ and Java projects and identify any correlation between software metrics and programmer's naming practices;

- **RQ$_4$: *What is the perception of software developers about the investigated naming categories?***

  We set out to probe into programmers' perceptions regarding the use and occurrence of the eight investigated naming practices.

3.2 Project Selection

Our sample comprises 40 open-source Java projects and 40 C++ projects hosted on GitHub. These projects are listed in Tables 1 and 2. We included widely used projects, most of which have been under development for at least five years (e.g., `fastjson`, `jenkins`, `junit4`, `mockito`, `retrofit`, `spring-boot`, `tomcat`, `pytorch`, and `tensorflow`). Also, some projects were taken into account because they appear in a curated list of "awesome" projects.[1] Table 1 and 2 give an overview of the examined projects. As shown in these tables, our Java and C++ samples cover somewhat small codebases (with less than 10K LoC) and large-scale ones (with over 100K LoC). Overall, we selected heterogeneous Java and C++ projects from a broad range of domains: e.g., software testing, game design, web applications development, image manipulation, and natural language processing. The selected projects also have a reasonable number of attributes, parameters, and variables names and were developed collaboratively by a diverse group of programmers. Therefore, we consider that we have selected a somewhat representative set of Java and C++ projects.

The Java projects were collected in July 2021 from GitHub by cloning and storing their respective repositories. In a similar fashion, we extracted the information from the selected C++ projects in January 2022. After storing the repositories, we extracted three common software metrics: (i) the total lines of code (we excluded non-functional code such as comments and white-spaces); (ii) the number of commits; and (iii) the number of contributors. To answer RQ$_3$, we correlated these metrics with the prevalence of the categories in projects.

[1] java-lang.github.io/awesome-java

3.3 Names Extraction

In order to extract identifier names from each project, we created a parser based on the SrcML tool (COLLARD, 2013). SrcML is a multi-language parsing tool for the analysis and manipulation of source code. SrcML turns source code into a document-oriented XML format, which allows for queries using XPath. For example, the srcML format contains structural information (markup tags) about identifier declarations (`<decl_stmt>`), associated types (`<type>`), and context (`<block>`).

We extracted 2,603,381 names from the 80 collected projects, and after applying the naming categorization, we get a total of 753,811 identifier names distributed across the categories (*Kings*, *Median*, *Ditto*, *Diminutive*, *Cognome*, *Index*, *Shorten*) as shown in Tables 1 and 2. The experimental package is available in Github [2].

To investigate and get an overview of the elements in the *Famed* category, we used the entire dataset extracted from both programming languages. We examined the name of each extracted identifier and the associated *Type* to answer RQ_1 and RQ_3. Therefore, for each naming category practice we report the occurrences in the studied projects and across them. To answer RQ_2 we analyzed the context where identifiers were declared.

3.4 Survey Design and Sampling

To answer *RQ*4 we designed an online questionnaire containing fifteen closed ended questions related to naming practices. A brief description (in Portuguese) and an example accompanied these questions (see Appendix A). We also included two initial questions to collect the demographic information of the respondents. The respondents had to point out their experience in software development as a single choice from four options: under two, two to five, six to 10, or over ten years; and also their education level (undergraduate, graduate, postgraduate). We selected the web-based questionnaire to conduct our survey because it maximizes the number of possible respondents. The Google Forms was chosen to host the questionnaire and enable data collection and pre-processing. The questionnaire was first trialed within the authors' organizations, with one of the authors registering possible observed issues. Some minor adjustments were made to ensure the consistency and clarity of the questions. Finally, the questionnaire link was posted to multiple websites (e.g., forums) and online groups (e.g., discord, whatsapp).

The overall purpose of this research is to study and analyze names chosen by developers to be used as identifiers, such as variables, attributes, method names and so on. We chose Java and C++, both object-oriented languages(OO), to be the composite the programming languages that are going to be studied. This choice was made on the

[2] github.com/rng-lab/naming-practices-analysis

idea that OO languages instigates the developer to chose descriptive names, because of the idea that OO languages represents ans abstraction of the problem that is trying to be solved. If the developer is programming an API with Java to create a payment systems, it is logical to chose names that are on board with this scenario. Probably the developer is going to use names such as "credit", "deposit" and "loan", because they are attached with the context provided by the abstraction of the problem's scenario. This motivation, alongside the fact that Java and C++ are broadly used in several situations and contexts, made us chose these two languages to composite our language arsenal.

This study is divided in two parts, the first consisting in calculating similarities between words in similar scopes, and the second one consists in defining name categories, and checking how these categories prevail in identifiers present in these projects, or "in the wild". The projects that we've chosen were used equally by both of the parts of this study, consisting in 40 Java projects and 40 C++ projects. In the direction of having better results, regarding different naming practices and the semantics of the words, we searched for projects with different domains of application. We thought that a large number of projects, arising from a diverse set of domains, would benefit the output of this research. These domains goes from a wide range, such as: NLP projects, editing and image manipulation projects, unit tests suites, computer vision projects, web frameworks, game design libraries and much more. We believe that this wide range of application domains could benefit our results and the quality of our study, limiting the influence of having a lot of projects regarding few domains. This would cause an negative impact, because a domain can have an direct impact on the names used by the developers, as we saw earlier, and this had to be diminished. For example, a certain domain could have a lot of specific words, that could not be possible to replicate to others domains, harming the result of the study. So, with 80 different projects from 2 OO programming languages, was possible to focus solely on the identifiers present in the projects, and not on their particular cases.

In total, we have chosen 80 different projects, divided equally between Java and C++ open-source projects. Regarding the size and impact of the projects, we opted to provide a wide range of sizes, including relatively small projects, with dozens of contributors and a few thousand commits, to considerable large projects, containing hundreds of contributors and hundred thousand commits. But all those projects, disregarding their size, are open-source projects, that are relevant in their respective domains. The table 1 illustrates all projects, alongside some information about them, such as: number of commits, number of contributors and quantity of lines of code. All of those projects mentioned above were discovered in a Github repository called "awesome-projects", in which displays several open-source repositories in different fields. Doing so, we believe that the quantity, size, different domains and relevance of the projects we've chosen contributed with the success of this study, and were more than enough to recall significant information about them.

3.5 Extraction of identifiers in source code

After choosing all 80 C++ and Java projects, the next step was to develop a tool capable of going through all files in the source code, analyzing them, identifying the identifiers, and storing them in a CSV file. Not only the names of those identifiers were relevant to our study, but also some information about them, such as their types, the class and/or method they were declared, and if they were declared within some scope such as inside a for loop, an if condition, or a while loop. This information was useful for defining naming categories, which are going to be displayed in the following sections. In total, 1.421.607 identifier names were collected in projects written in Java, and 1,181,774 identifiers were collected in C++ projects.

In the first study, the meaning of identifier names was compared between them, respecting their scope and class. Hence, it is essential to extract not only the attributes or variables but also the name of the class and method in which they were declared. The tool we have developed recognizes the declaration of an identifier and stores it in a CSV containing these respective fields: name, type, scope, class, method (if it is variable), filename, and project. With this information, we could link an identifier with the context in which they were found. Collecting the type of identifier enabled the insight into some naming categories.

The first step of the development process was to use a tool to ease the finding of the identifier names present in the projects. We used a tool that made the process of verifying if a word in the file was an identifier or other elements present in the source code, such as reserved words and imports. To achieve that, we used a tool called SrcML, which converts the source code into an XML (Extensible Markup Language) file. This XML file represents aspects of the source code in the form of tags, including identifier declarations, that made it possible to analyze the source code more efficiently.

3.5.1 SrcML

As mentioned, a tool named SrcML was used to assist in the process of collecting identifier names from the source code that was found. With SrcML is possible to convert Java or C++ files from a program to its own XML version, making it easier to identify those identifiers. Alongside Java and C++, there are other programming languages supported by SrcML, with few differences between them, such as specific tags for specific language commands.

The source code is represented by markup tags presented in an XML file, which every reserved word or command has its own tag, and respects the hierarchy contained in the original file. In a Java file, for instance, there is a root tag, representing the original class, and several other tags, such as tags for methods and attributes, that are considered

children from the root tag. Markup tags in a SrcML file use the XML tree structure to represent scope nuances from a Java or C++ file. Tags present in the same hierarchy give context to another tag, such as the tag "name", which describes the name of the identifier, or the name of its type. For example, first, we have a tag representing a class, followed by a tag corresponding to its name and a specifier tag, explicating if a class is private, public, etc. A SrcML file preserves all source code from the original file, including comments, formatting, and preprocessor directives, allowing us to easily receive information about the original file.

The SrcML source code representation using markup tags facilitates the structural vision of the code itself and enables the usage of queries to discover insightful information about the code. Xpath queries are used to achieve this objective, for instance, a XPath query can be used to discover identifiers that are declared within a certain method or code location. With the aid of these queries, it could be possible to determine the scope of identifiers with information such as their type. Because the SrcML file is an element tree, it is easy to perceive different scopes in the original code to the hierarchy in an XML file. If an identifier is declared within a "if" tag, the tag that represents the identifier is a child of the "if" tag, and could be found using XPath queries.

3.5.2 Identifying Identifier Names

In order to make those SrcML file useful, a tool was developed to be capable of reading and understanding XML logic and syntax to extract identifiers correctly. Not all markup tags contain relevant information for our study, and to access those that are indeed relevant, our tool must the capable of following the logic behind an XML file, including the distinction between different levels of the tree structure, and identify the names of the tags that carry relevant information. Our tool is based on several XPath queries, recognizing code locations in which identifiers were declared.

The first thing our tool was designed to do is perform an XPath query that returns the content of all classes present in a project. This action will leverage our study with information about classes themselves, and the quality of identifiers declared within the class. Later in this study it will be displayed a level of naming quality based on each class a project had. After retrieving the content of a class, another XPath query collects all of its attributes and types. If there are methods inside the class, the variables declared within the method are collected in an analog way. The last step is to perform an XPath query for every programming command, and in the same way, identify and collect variables declared in those scopes. With that logic is possible to gain information if there is any difference in naming quality in different scopes in the code. Performing a different XPath query for classes, methods, and commands allows our tool to correctly assign an identifier to a class or method.

3.6 Naming Practice Categories

The categories presented in this subsection are a compilation of programmers' practices reported in several studies (ARNAOUDOVA, 2016),(BENIAMINI, 2017) (AL-SUHAIBANI, 2021), and books (MARTIN, 2008),(DILEO, 2019). Inspired by antipattern templates (BROWN, 1998), in order to explain the naming practice categories, we frame the discussion of each category in terms of the following elements: category name, examples, motivation (why), consequences of the naming practice, and recommendations. All the categories were created exclusively for this study, with the purpose of identifying common bad naming practices and summarizing them into different categories. We design each category by first identifying the cause of misunderstanding in identifier naming; for example, identifiers consistent with single letters can produce misinformation. Hence, we proceed by adding examples, consequences, and recommendations to the category. The categories involve the practice of using numbers somewhere in the name, including the length of the name and its relation to its type. Using numbers in an identifier name, depending on the context, can be harmful to its pronunciation and comprehension (MARTIN, 2008). The categories that consider the length of the identifier are based on the state that shorter identifiers take a longer time to be understood than bigger ones (HOFMEISTER, 2017). Finally, using the type's name in the identifier name is a practice that is usually recommended in IDE's (Integrated Development Environment) (DEISSENBOECK; PIZKA, 2006), and can also possibly prejudice code's comprehension by not aiding the intention behind the naming, and confuse the reader with the addition of noise words.

Listing 3.1 shows an actual code snippet from a Java project used in this study, which are some "in the wild" examples of naming categories. First, the method name "getUriBuilderWithoutQueryParams" is an extension of its type "UriBuilder." Notice that the information about the method being related to the Uri building has already been told by the type, and adding it to the identifier increases its complexity needlessly. The method signature can inform the reader by combining the type and the name without the need for repeating words. The first variable in the snippet also has a problem involving its type: they are the same. The identifier called "partialUpdateEntityRequest" is identical to its type, and it does not create any new information for the reader. This name lacks intention behind its naming, and it is solely a copy of the type's name. The second variable is called "b". The problem with that name is also about the need for more intention; the name is only a single character; it does not represent the intention behind the naming and, additionally, does not give any information to the reader, possibly causing misinformation.

Listing 3.1: Java code Example

```
public UriBuilder getUriBuilderWithoutQueryParams(String[] args)
{
    PartialUpdateEntityRequest<?> partialUpdateEntityRequest = getReq()
    UriBuilder b = super.getUriBuilderWithoutQueryParams()
    appendKeyToPath(b,partialUpdateEntityRequest.getId() )
}
```

3.6.0.1 Kings

This category represents identifier names composed by numbers at the end. The name Kings comes from adding a number at the end of a King's name to differentiate it from others with the same name. That is the logic behind the King's category; it was based on developers that use numbers to differentiate identifiers. This practice precludes a more intentional revealing nomination between identifiers because the one thing that makes the names different is a number. **Example**: `String name1` and `String name2` or `Integer arg1` and `Integer arg2` represent arbitrary distinctions as number series. **Why**: programmers often opt to employ names that fall into this category to distinguish between identifiers that appear in the same scope. **Consequences**: names with numbers at the end, however, are not very informative and do not represent intentional naming (MARTIN, 2008),(DILEO, 2019) . **Recommendation**: usually, identifiers represent different things; whenever that is the case, they should be named accordingly (MARTIN, 2008).

3.6.0.2 Median

This category is a variation of the *Kings* category and comprises identifier names composed of numbers in the middle. **Example**: the names `fastUInt64ToBuffer` and `base64Bytes` contain numbers that might be representing 64 bits values. **Why**: numbers in the middle, in general, are used to denote the value stored in the attribute/variable or even to provide some distinction among similar identifier names. **Consequences**: names with numbers in the middle can potentially be harder to search for in the source code, hard to pronounce, and also can be very similar to other names that differ only in terms of the numbers that appear somewhere in the middle (MARTIN, 2008). **Recommendations**: programmers should use numbers only when necessary and surround numbers with pronounceable words (MARTIN, 2008).

3.6.0.3 Ditto

The category *Ditto* consists of identifier names spelled in the same way as their *Types*. **Example**: `timeZone` is spelled as its *Type* `TimeZone` in the same way that the name `object` has the same name as its *Type* (`Object`). **Why**: naming identifiers according to the respective type is an easy option to avoid mental mapping (which usually are associated

with the problem domain concepts). **Consequences**: this naming practice might result in names that are harder to map to their purposes when used in larger scopes, and tend to cause misinformation when the type name changes but the identifier names do not (MARTIN, 2008),(ALSUHAIBANI, 2021). **Recommendations**: avoid using *Ditto* based names in very large scopes and/or in contexts in which other names can conflict with them (MARTIN, 2008).

3.6.0.4 Diminutive

This category includes identifier names that are a derived from segments of their respective *Type* names. **Example**: `listener` is an example of a name in this category when its associated *Type* is named *EngineTestListener*. The name `NFRuleSet ruleSet` is also considered as a chunk of its *Type*. **Why**: developers usually rely on short names to avoid overloading the reader with many concepts. **Consequences**: when used in large-scope contexts, names that fall into this category might impair code comprehension (MARTIN, 2008). **Recommendations**: programmers should use names that properly convey the identifier's purpose within the local context and scope (MARTIN, 2008).

3.6.0.5 Cognome

Identifier names in this category contain as an additional suffix or prefix the name of the respective *Type*. **Example**: an identifier `nameString` includes in its name the the respective *Type* name (`String`). **Why**: usually programmers resort to adding suffixes in names to help them remember the *Types*. **Consequences**: encoding *Type* into names might place an extraneous cognitive load on the programmer (MARTIN, 2008; DILEO, 2019). **Recommendations**: give identifiers names that are meaningful without having to resort to adding its *Type* information to the names (MARTIN, 2008).

3.6.0.6 Index and Shorten

These categories represent similar naming practices: naming an identifier with a single-letter word. The *Index* category represents names with one arbitrary letter. Names in the *Shorten* category are the starting letters that correspond to their respective *Types*. **Example**: the names `Integer i` and `Integer j` falls into the *Index* category and `Person p` and `String s` are examples of *Shorten* names. **Why**: single-letter names are traditionally used to identify counters in loops. **Consequences**: single-letter names usually are not easy to locate in the source code (unsearchable) and, when employed in large scopes, can be hard to be understood (MARTIN, 2008),(DILEO, 2019),(BENIAMINI, 2017). **Recommendations**: use single-letter names only in local and small scopes; otherwise, intent-revealing names are better (MARTIN, 2008).

3.6.0.7 Famed

This category includes very common names; that is, when naming become arbitrary and programmers need to come up convenient defaults. *Famed* names appear in almost every source code, potentially, in similar contexts, such as in loop statements (e.g., FOR). **Example**: the word `i` is a recurrent identifier name used in loops to denote counters. **Why**: very popular identifiers are part of the programmer mindset and can be quickly remembered and understood. **Consequences**: when used in an indiscriminate fashion, they may cause misinformation (MARTIN, 2008; ALSUHAIBANI, 2021). **Recommendations**: use intent-revealing names even in short-scope contexts (MARTIN, 2008; ALSUHAIBANI, 2021).

3.7 Results

In this section, we present the results of our empirical study around the RQs described in the previous sections.

Table 3: The top 10 names in *Ditto* category

Names	Num. Repetitions	Num. Projects
Ditto in Java programs		
`url`	2,421	24
`list`	1,464	32
`file`	1,444	32
`method`	1,044	29
`context`	1,042	25
`object`	991	29
`uri`	968	25
`node`	844	21
`type`	593	30
`date`	526	25
Ditto in C++ programs		
`T`	1,227	34
`string`	1,134	18
`uint8_t`	564	15
`args`	247	22
`t`	231	20
`std`	143	19
`type`	141	19
`handle`	96	17
`mode`	45	16

Table 4: The most common names (*Famed*)

Names	Num. Repetitions	Num. Projects	Common Type	Num. Occurrences	Num. Different *Types*
Famed in Java programs					
`value`	16,940	40	`String`	3,345	598
`result`	12,975	39	`int`	1,924	887
`name`	11,374	40	`String`	10,208	116
`i`	11,172	39	`int`	9,794	139
`e`	10,225	40	`Throwable`	1,851	589
`index`	8,224	38	`int`	7,184	83
`key`	7,696	35	`String`	3,187	205
`s`	7,442	35	`String`	2,771	318
`c`	7,337	35	`int`	1,468	441
`t`	6,989	37	`Throwable`	1,210	336
`a`	6,970	34	`float`	739	575
`b`	6,511	38	`int`	983	486
`type`	6,162	40	`Class`	1,523	315
`input`	6,008	37	`String`	565	277
`p`	5,256	35	`int`	381	443
`source`	5,025	37	`String`	765	263
`n`	5,010	34	`int`	2,930	165
`request`	4,719	32	`Request`	1,489	212
`context`	4,437	37	`Context`	1,042	241
`id`	4,216	36	`String`	1,523	104
Famed in C++ programs					
`i`	5,421	40	`int`	2,362	151
`value`	3,912	40	`double`	427	268
`x`	3,856	36	`double`	858	250
`result`	3,771	40	`T`	448	231
`index`	3,106	38	`int`	869	88
`n`	3,027	37	`int`	729	159
`ctx`	2,964	22	`OpKernelConstruction`	622	105
`name`	2,545	37	`string`	950	187
`type`	2,534	40	`int`	306	426
`b`	2,370	39	`bool`	386	219
`p`	2,351	37	`void*`	190	412
`size`	2,285	39	`size_t`	619	119
`context`	2,279	34	`OpKernelConstruction`	501	133
`s`	2,254	35	`Status`	427	243
`len`	2,101	34	`Uint32`	463	47
`node`	2,093	30	`Node`	154	286
`v`	1,983	38	`double`	118	253
`data`	1,832	37	`void*`	441	211
`val`	1,821	35	`int`	192	199
`c`	1,776	38	`char`	246	199

3.7.1 RQ_1: How prevalent are the naming practice categories?

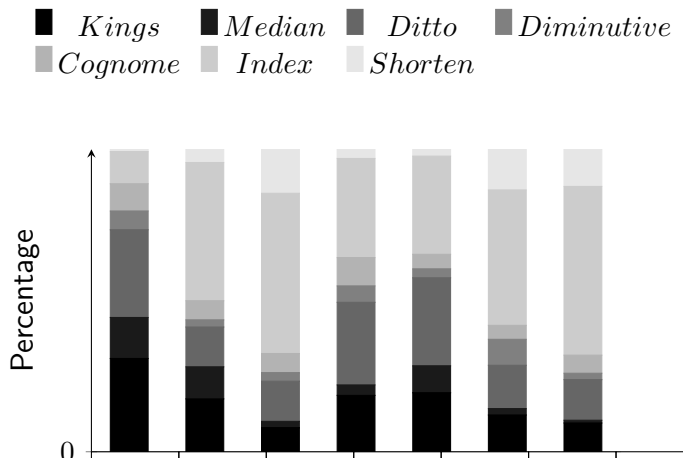
To answer RQ_1, we analyzed the categories *Kings*, *Median*, *Ditto*, *Diminutive*, *Cognome*, *Shorten*, and *Index* regarding how commonly they appear in the projects in our samples. Tables 1 and 2 list how common each of these categories are across the 80 investigated projects. Considering the identifier names in the chosen Java projects, 20.79% are composed by numbers at the end (*Kings*), 7.65% have numbers in their middle (*Median*), 26.24% are spelled the same as their *Types* (*Ditto*), 7.50% contain the hole *Types* as a sub-part (*Cognome*), 4.51% have in their spelling a sub-part of their respective *Types* (*Diminutive*), 3.35% are single-letter names composed of the first letter of their *Types* (*Shorten*), and 29.93% are arbitrary single-letter names (*Index*). As for the C++ projects in our sample, only approximately 7.28% of the identifier names fall into the *Kings* category, 53.24% of the identifiers are named according to their respective types (*Ditto*), around 9% follow the *Cognome* naming practice, 11.86% of the C++ identifier names are *Diminutive*, only 2.3% belong to the *Shorten* category, and approximately 13% of the C++ identifier names are single-letter names (*Index*).

These results indicate that the use of single-letter names (*Index*) is a widespread naming practice adopted in object-oriented programming. Indeed, (BENIAMINI, 2017) have observed that single-letter names account for 9–20% of names in Java programs. As stated by them, the most commonly occurring single-letter name is `i`, and in some cases, `j` is also highly used. In addition, we observed that single-letter names representing contractions of their respective *Type* are not so common (*Shorten*), but are prevalent across projects (see Section 3.7.3). Programmers seem to be conscious about single-letter names implications (HOFMEISTER, 2017), and thus avoid choosing such naming practice: this category represents only 3.35% (14,088) of the examined Java names and 2.3% (7,689) of the identifier names in C++ projects.

Names that fall into the *Ditto* naming practice category make up the lion's share of all identifier names in C++ (53.24%) projects and are the second most common naming practice in Java (26.24%) programs. Even though it might be argued that *Ditto* is a sound naming practice given that it leads to pronounceable names and many IDEs suggest names that include the identifier *Type*, in most cases, the practice does not lead to the creation of intention-revealing names.

Table 3 lists the five most reoccurring names in such a category for Java and C++ projects. According to Table 3, the use of identifier names as `list`, `object`, `args`, `unit8_t` and `t` are common, but these names do not reveal intentions. When the context is not explicit or broad, programmers have to trace back what kinds of data are in an identifier named as `list` or `t`. These names are generic and hurt the reader's understanding. Moreover, whether the *Type* name changes, then the identifier names will be misleading as in cases such as `string` and `type`. According to (AVIDAN; FEITELSON, 2017), the

Figure 1: Naming practices distribution over Java programming statements



	Attr	For	If	Method	Param	Switch	While
Kings	30.84%	17.49%	7.98%	18.64%	19.53%	12.16%	9.51%
Median	13.56%	10.60%	2.07%	3.46%	8.90%	2.11%	0.91%
Ditto	29.01%	13.15%	13.28%	27.31%	29.10%	14.32%	13.43%
Diminutive	6.20%	2.38%	2.84%	5.43%	2.95%	8.59%	2.11%
Cognome	9.01%	6.32%	6.31%	9.36%	4.81%	4.62%	5.98%
Index	10.63%	45.70%	52.99%	32.78%	32.46%	44.77%	55.79%
Shorten	0.76%	4.36%	14.53%	3.04%	2.24%	13.42%	12.27%

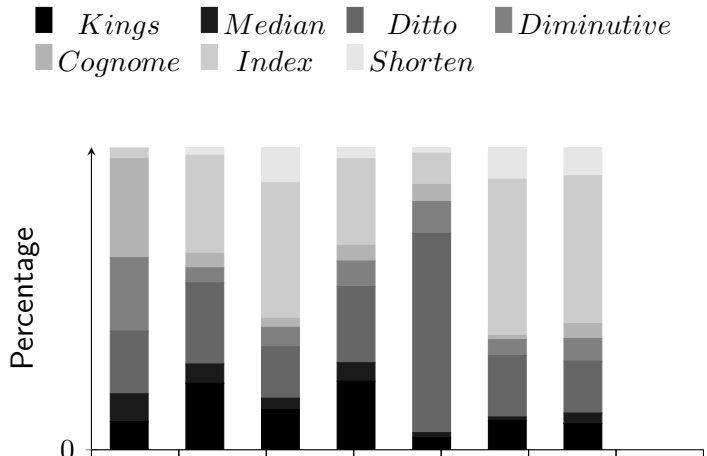
Statements

evil face of names is misleading names.

The habit of choosing names that represent arbitrary sequential distinctions also revealed a common practice among Java and C++ programmers (*Kings*). However, number-series is considered a bad practice in object-oriented programming when creating meaningful names. Number-series naming is a non-informative option, which might disturb code comprehension and maintainability. The use of numbers in the middle of names, although prevailing in the studied names, does not appear to be a recurrent naming practice. We observed that the most common numbers used in the middle of names are: (i) 0, 1, 2, 3, 4, 5, and 6 – as well as meaning some distinction; and (ii) 8, 16, 32 and 64 – meaning identifiers which might be representing 8, 16, 32 or 64 bits values, respectively.

The scenarios in which programmers choose names that are variants of their *Type* are also common. For example, names that contain sub-parts of their *Type* (*Cognome*) account for 7.50% of the identifier names in Java projects and around 9% in C++ programs. Often, these identifier names represent prefix/suffix (noise words) conventions, such as: `streetString`; `listPersons`; `floatArg`. Noise words are redundant and should never

Figure 2: Naming practices distribution over C++ programming statements



	Attr	For	If	Method	Param	Switch	While
Kings	9.34%	21.98%	13.31%	22.64%	3.98%	9.59%	8.71%
Median	9.16%	6.39%	3.75%	6.14%	1.68%	1.26%	3.42%
Ditto	20.75%	26.81%	17.01%	25.17%	65.86%	20.25%	17.17%
Diminutive	24.27%	4.96%	6.42%	8.44%	10.54%	5.33%	7.49%
Cognome	32.68%	4.78%	2.97%	5.18%	5.65%	1.26%	4.88%
Index	3.41%	32.42%	44.80%	28.61%	10.32%	51.74%	48.90%
Shorten	0.38%	2.65%	11.75%	3.81%	1.97%	10.56%	9.44%

Statements

appear in names. In general, `streetString` is not better than `street`. Short names are in general easier to comprehend and one of the first things a programmer can do to keep identifier names short is to avoid adding unnecessary information. In contrast, names that are part of their *Type* are not so common. These names are hard to search for and are not very meaningful in most contexts.

Table 5: Spearman correlation

Category	LoC				Commits				Commiters			
	Java		C++		Java		C++		Java		C++	
	Corr	p-value	Corr	p-value	Corr	p-value	Corr	p-value	Corr	p-value	Corr	p-value
Kings	0.337	0.038	0.391	0.014	0.150	0.365	0.199	0.222	0.053	0.748	0.090	0.583
Median	0.254	0.123	0.004	0.978	0.054	0.743	-0.197	0.226	-0.081	0.627	0.070	0.668
Ditto	-0.517	0.001	-0.049	0.763	-0.216	0.191	0.074	0.649	0.101	0.545	-0.041	0.801
Diminutive	-0.021	0.898	0.335	0.037	0.008	0.959	0.225	0.166	-0.171	0.304	-0.025	0.875
Cognome	-0.227	0.169	0.268	0.098	-0.300	0.066	0.188	0.250	-0.178	0.283	-0.103	0.532
Index	0.341	0.036	-0.330	0.040	0.133	0.421	-0.311	0.054	-0.098	0.554	0.010	0.950
Shorten	0.387	0.016	-0.196	0.229	0.124	0.453	-0.110	0.501	-0.068	0.681	0.128	0.435

3.7.1.1 Very Common Names

In a study about how developers choose names, the authors observed that the probability of two programmers choosing the same name is low: the median probability was only 6.9% (FEITELSON, 2020). At the same time, when a specific name is chosen, it is usually understood and often used by most programmers (AVIDAN; FEITELSON, 2017),(SWIDAN, 2017). In fact, we observed that there are some frequently used names. The Top-3 most common names in Java programs are (see Table 4): (i) `value` (16,940 occurrences); (ii) `result` (12,975 occurrences); and (iii) `name` (11,374 occurrences). It might be expected that `i` is a widespread name (BENIAMINI, 2017), but many other single letter names are also commonly used across Java projects (e.g., `e`, `s`, `c`, `t`, `a`, `b`, `p`, `n`). Most of them are in the Top-10 most common names. Another interesting observation is `index` and `key` as part of the Top-10 most common names. Overall, some of the common identifier names in Table 4 are popular in programmer's vocabulary: `value`, `result`, `name`, `index`, `key`, `type`, `input`, `source`, `request`, `context`, `id`.

As for C++ programs, the three most common identifier names are (i) `i` (5,421 occurrences), (ii) `value` (3,912 occurrences), and (iii) `x` (3,856 occurrences). According to our results, many of the identifier names shown in Table 4 are widely common in programs written in Java and C++: `value`, `result`, `name`, `index`, `type`, `context`, `i`, `b`, `n`, `p`, and `s`. It turns out that `value` appears among the top three most used identifier names both in Java and C++. Java programmers seem to have a slight preference for the names `result` and `name` in comparison to C++ programmers. As mentioned, some single-letter names are widely used by programmers in both languages, being `i` the most commonly used

single-letter name in Java and C++.

Further analysis of the names in Table 4 and their corresponding most common *Types* led to interesting results about programmers' rationale when programming in Java and C++. As noted by (BENIAMINI, 2017), analyzing this link yields interesting results because it is possible to understand the meaning related to names frequently used by programmers, especially single-letter names. We can observe most identifier names are associated with `int` variables (e.g., `result`, `i`, `index`, `c`, `b`, `p`, `n`) or `String` *Types* (e.g., `value`, `name`, `key`, `s`, `input`, `source`, `id`). As shown in a survey conducted by (BENIAMINI, 2017), single-letter names such as `i` and `j` are understood as counter variables (integer values) and most of the time used as loop control variables.

There are other interesting findings. For example, in Java programs the single-letter name `e`, is usually correlated with error and exception (BENIAMINI, 2017). Our results show that `e` is mainly associated with the `Throwable` *Type*. In the same way, `s` is a single-letter name essentially associated with `String` (see Table 4). However, we also found some counter-intuitive results. For instance, contrary to our expectations, we observed that in programs written in Java the single-letter name `b` is not linked with boolean values (BENIAMINI, 2017) but with integer values. Additionally, the identifier name `t` is mainly associated with `Throwable`; which is somewhat counter-intuitive because `t` is also often used to name and convey the idea of time-related constant values and variables or variables that hold temporary values (BENIAMINI, 2017).

Other names that seem to have meaningful associations are the following: `type`, which is generally associated with the `Class` *Type*; `context` and `request`, which are often associated with the `Context` and `Request` *Types*.

Our results would seem to suggest that the underlying meaning of the identifier names vary a lot. For example, the name `result` was associated with 855 different *Types*. The name `i`, which intuitively is associated with index (`int`), also assumes other 139 different *Types*. Nevertheless, in most cases (9,794 out of 11,172), this name is associated with integer values. The name `name` seems to be usually associated with the `String` *Type*: 10,208 out of 11,374 occurrences are associated with `String`.

3.7.2 RQ_2: Are there context-specific naming practices categories?

To answer the RQ_2, we investigated the predominance of the naming practice categories over particular contexts (ATTRIBUTE, PARAMETER, METHOD, FOR, WHILE, IF, and SWITCH). The results are present in Figure 1 and 2.

We found that while some naming practices (ALLAMANIS, 2014) acknowledge the use of single-letter words (*Index* and *Shorten*) to name a local, temporary or loop variable, this practice is much more pervasive than any other. Except for naming attributes

Java and C++, in which case Java programmers prioritize the use of *Ditto* and *Kings* naming practices while C++ programmers tend to use *Cognome*, *Ditto*, and *Diminutive*. Surprisingly, names with numbers at the end appear 30,655 times in our study as Java attributes and only 4,066 in class attributes in C++ projects. Especially in large-scope contexts, *Kings* names should always be avoided by programmers. In contrast, using *Ditto* names in such a case seems to be a reasonable choice. IDEs (e.g., Eclipse and Intellij IDEA) usually analyze the scope and generate suggestions from the current context and these suggestions often include information regarding the respective *Type*.

Focusing on particular contexts, we might see that programmer's practices are context-specific. For example, the use of practices that might result in meaningful names (e.g., *Ditto*) is more common in long-scope contexts (ATTRIBUTE and METHOD) than in short-scope ones (IF, FOR, WHILE, SWITCH). Especially in C++ projects, *Ditto* makes up for the lion's share of the parameters names. Java and C++ programmers seem to adopt less descriptive names in the context of `switch` and `while` statements. As shown in Figures 1 and 2, *Index* names appear more often inside contexts surrounded by `if`, `for`, `switch`, and `while` statements, where their occurrence is widely and accepted (KERNIGHAN; PIKE, 1999; BENIAMINI, 2017). However, as observed by (AVIDAN; FEITELSON, 2017), hiding the plural names using single-letter words may camouflage the meaning of the respective identifier. It might not be a natural interpretation that the identifier stores more than one object.

The predominance of *Kings* and *Index* as parameter names do not agree with the findings of (AVIDAN; FEITELSON, 2017). Their experiment indicated that parameter names contribute more to code comprehension than any other names (e.g., attributes or local variables). Since parameters are part of the method header and the starting point of the comprehension task, programmers pay special attention to parameter names in order to better understand the method behavior (AVIDAN; FEITELSON, 2017). However, every naming practice category we studied are used to name parameters, although, as observed by (AVIDAN; FEITELSON, 2017), parameter names are often more carefully chosen by programmers.

3.7.3 RQ_3: Do the naming practice categories carry over across different Java and C++ projects?

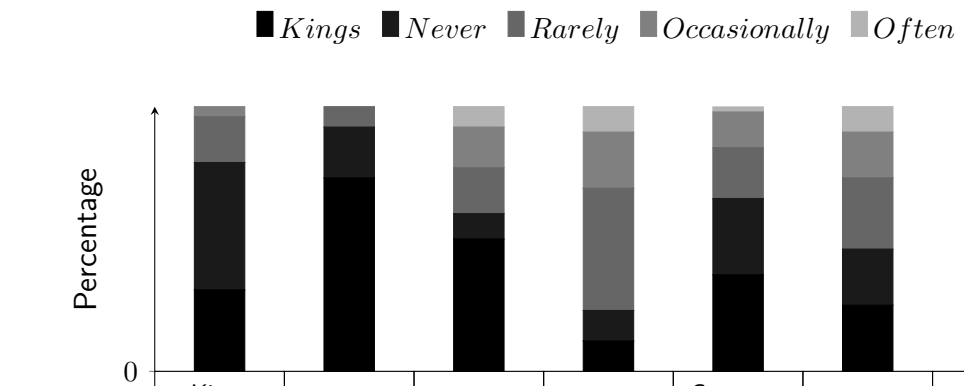
In hopes of answering the RQ_3, we analyzed the prevalence of naming practice spanning multiple projects. Tables 1 and 2 list the categories by projects. All selected projects turned out to have problematic names, which suggests that the investigated naming category practices are probably not uncommon. Even the most popular projects have naming practices which might result in meaningfulness names (e.g., `fastjson`, `jenkins`, `junit4`, `mockito`, `retrofit`, `spring-boot`, `tomcat`, `tensorflow`, and `pytorch`).

As highlighted in Tables 1 and 2, *Ditto* and *Index* are very common naming practices. Especially, these practices are dominant (representing more than 50% of analyzed identifiers) in some projects. For example, *Ditto* names are widely used in Java and C++ programs, accounting for 85.08% in `riptide` (Java), 80.78% of the identifier names in `clickhouse` (C++), 78.26% in `webmagic` (Java), 72.93% of the names in `kdenlive` (C++), 68.60% in `mysql-server` (C++), 68.32% in `percona-server` (C++), 65.99% in `keywhiz`, and 54.46% in `aeron`. The problem with *Ditto* is that when the *Type* changes, the identifier name might lose its meaning (SCALABRINO, 2017). *Index* names appear to be more common in Java programs. For instance, these identifier names account for 58.93% of all identifiers in `boofcv` (Java) and 66.53% in `rxjava` (Java). It would seem that *Index* names are not very common in C++: `proxysql` which is the program in which *Index* names are most common, has around 34.7% of the identifier names following this naming practice. `rocksdb` and `citra` also include a substantial amount of identifiers named according to the *Index* naming practice: 34.71% and 30.30%, respectively.

In some isolated cases, some name practice seems to be dominant, as *Kings* in `fastjson` (49.88%) and `libgdx` (47.83%). On the other hand, the naming practices *Cognome*, *Diminutive* and *Shorten* are not dominant in any specific project. Specifically, *Shorten* seems to be a naming practice that most programmers try to avoid: programmers avoid naming identifiers using the first letter of the *Type*. As mentioned, *Shorten* names usually are not easy to search for in the source code and, when employed in large-scope contexts, they tend to be hard to understand.

To better comprehend whether the project's characteristics may influence the prevalence of one practice, we looked at the correlation between common software metrics (e.g., lines of code, number of contributors, and number of commits) and the predominance of the naming practice categories. Table 5 summarizes the Spearman test results. The results show no representative correlation between the investigated project characteristics and the categories of naming practices. Overall, we can observe a low correlation between the number of contributors and the prevalence of any category. One might surmise that an increase in the number of programmers might be beneficial towards removing bad naming practices. However, this does not seems to be the case. The same rationale might be employed to the number of commits: whether the project evolves, the quality of the identifiers names might evolve or decay. Though, in contrast to (DEISSENBOECK; PIZKA, 2006), which stated that identifiers names are subject to decay during software evolution, the results show that it might not seem to be the case. Especially observing LoC, we might observe some compelling correlations. For example, there is a negative correlation (rho -0.517) between size and the category *Ditto* (for Java programs). Therefore, names spelled in the same way as their respective *Types* tend to be way more common in small projects. On the other hand, large Java projects might tend to contain names involving practices such as *Index* (rho 0.341) and *Shorten* (rho 0.387).

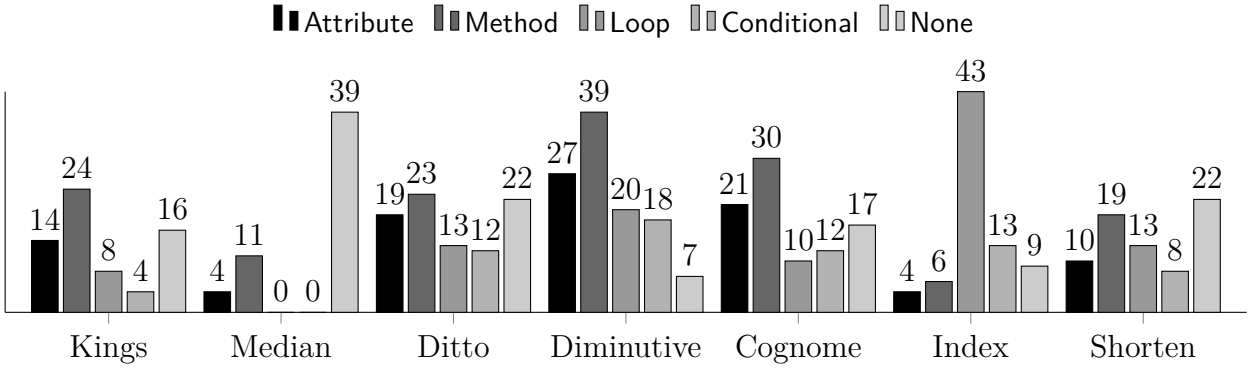
Figure 3: Naming practices distribution over programming statements



	Kings	Median	Ditto	Diminutive	Cognome	Index	Shorten
Never	30.8%	73.1%	50.0%	11.5%	36.5%	25.0%	40.4%
Rarely	48.1%	19.2%	9.6%	11.5%	28.8%	21.2%	26.9%
Occasionally	17.3%	7.7%	17.3%	46.2%	19.2%	26.9%	19.2%
Often	3.8%	0.0%	15.4%	21.2%	13.5%	17.3%	13.5%
VeryOften	0.0%	0.0%	7.7%	9.6%	1.9%	9.6%	0.0%

As shown in Table 1, *Ditto* and *Index* are the most dominant practice across Java projects. Considering only the two categories, they account for 235,886 identifier names, representing 56.17% of all analyzed names in Java projects. These results are consistent with the findings of (BENIAMINI, 2017). Although code conventions and style guides may constrain identifier naming practices, programmers seem to be heavily influenced by IDEs content assist capabilities. As programmers work in the editor, content assist analyzes their code and recommended elements to complete partially entered statements. Therefore, it is indispensable to provide more sophisticated and context-aware capabilities to assist programmers in naming and renaming identifiers (JIANG, 2019; ISOBE; TAMADA, 2018; PERUMA, 2018; PERUMA, 2019). Finally, programmers would seem to prioritize single-letters names in contexts where they are widely accepted (see Section 3.7.2).

Figure 4: Naming practices distribution over programming statements



3.7.4 RQ_4: What is the perception of software developers about the investigated naming categories?

This section presents the results of our survey with 52 programmers. We start by characterizing the respondents (Section 3.7.4.1). Next, we assess the relevance of the naming practice categories by how often they are used by programmers (Section 3.7.4.2). We then analyze how naming practice categories adoption varies according to programming statements (Section 3.7.4.3).

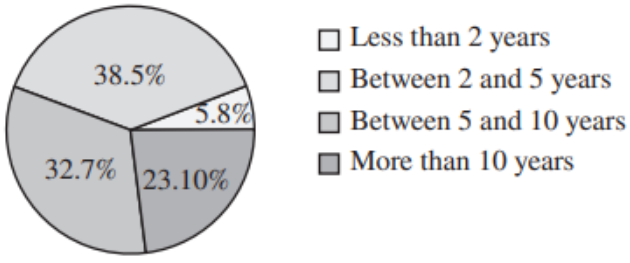
3.7.4.1 Respondents' Demographics

Figure 5 depicts the respondents' experience in software development and the corresponding frequencies and percentages. A total of 5.8% of the respondents have less than two years of experience, while 55.8% have more than five years of experience, suggesting that most survey respondents are experienced programmers. Moreover, we seem to have collected a reasonably balanced distribution of programmers in terms of education level. Figure 3.7.4.1 shows the respondents' education level. As the majority of the respondents (73%) have a graduate degree, we claim that it increases our confidence in the validity of the responses.

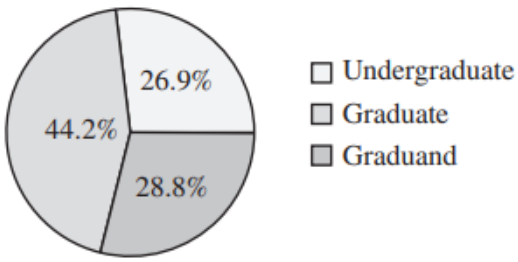
3.7.4.2 Most Commonly Used Naming Practices

The respondents were queried about how often they choose identifier names conforming to the naming practice categories. A five-point Likert scale was used to capture respondent opinions ranging from "Never" to "Very Often". Figure 3 shows how frequently respondents have been using each naming practice category. In our sample, *Diminutive* is

Figure 5: Respondents Demographics



(a) Respondents experience in software development



(b) Respondents education level

the most frequently used naming practice category (i.e., used "Often" or "Very Often"), followed by *Index* and *Ditto*. This result seems to align well with our observation about the prevalence of the naming practices in open-source object-oriented programming.

Notably, from the survey, we can make the following observations:

- All the respondents adopt at least one naming practice category "Occasionally" or "Often", with 26% (13) of the respondents claiming to adopt at least one naming practice "Very Often".
- *Diminutive* is the most adopted naming category practice by respondent. However, as we could observe, this naming practice category is not so prevalent in the analyzed object-oriented projects as claimed by the survey programmers.
- *Median* is the least adopted naming practice category (see Figure 3), with just 26% (14) of the respondents using it "Rarely" or "Occasionally". The lower use of this naming practice corroborates our observation that programmers seem to be conscious of this harmful practice in object-oriented programming.
- *Ditto* is not a widespread naming practice among the survey respondents. Only 12 out of 52 programmers (23%) indicated a tendency to write identifier names spelled in the same way as their *Types*; which do not ratify our previous observations about the prevalence of *Ditto* across Java and C++ projects (see Section 3.7.3). This contrasting result suggests that programmers might be not aware of their general use of naming practices. Moreover, this might also be a sign that naming assistant features present in modern IDEs do not influence the respondents.

3.7.4.3 Most Commonly Used Naming Practices According to Context

In order to specify the location in which programmers mainly observe the occurrences of the naming practice categories, the respondents were allowed to select multiple locations (ATTRIBUTE, METHOD, LOOP, CONDITIONAL, and NONE). This is expected to be done by remembering instances of naming practice categories encountered by respondents in their software development works.

The two most common answers from the respondents were: ATTRIBUTE and METHOD (see Figure 4). These findings share similarities with those presented in Section 3.7.2, wherein 56% of the names occur as ATTRIBUTE or are declared in the context of METHOD. One notary exception is *Index*, in which case, 43 out of 52 respondents indicated that this naming practice occurs mainly inside contexts surrounded by LOOP statements (FOR or WHILE). Indeed, as observed by (BENIAMINI, 2017), single-letter names can be used safely in a short-scope context. Finally, as expected, the majority of respondents (39 out of 52) indicated that they usually do not observe *Median* in their day-life (see Figure 4).

Table 1: Java programs used in our experiment.

Project	LoC	Contributors	Commits	Kings		Median		Ditto		Cognome		Diminutive		Shorten		Index		Total
				Total	%	Total.	%	Total.	%	Total.	%	Total.	%	Total.	%	Total.	%	
aeron	108,442	86	14,409	606	6.34	450	4.71	5,205	54.46	933	9.76	1,932	20.21	114	1.19	318	3.33	9,558
androidutilcode	39,030	32	1,317	179	7.74	21	0.91	1,170	50.56	385	16.64	73	3.15	77	3.33	409	17.68	2,314
archunit	100,276	49	1,499	91	3.07	16	0.54	1,744	58.86	596	20.11	303	10.23	9	0.30	204	6.88	2,963
boofcv	650,019	14	4,520	7,483	23.19	1,696	5.26	1,573	4.87	266	0.82	880	2.73	1,354	4.20	19,017	58.93	32,269
butterknife	13,279	97	1,016	135	21.95	8	1.30	358	58.21	68	11.06	14	2.28	4	0.65	28	4.55	615
corenlp	581,374	107	16,280	2,372	9.53	831	3.34	4,281	17.20	3,864	15.52	610	2.45	1,622	6.52	11,310	45.44	24,890
dropwizard	74,215	364	5,789	53	1.85	14	0.49	1,993	69.64	343	11.98	269	9.40	29	1.01	161	5.63	2,862
dubbo	179,477	386	4,681	754	6.39	81	0.69	6,983	59.19	1,096	9.29	644	5.46	369	3.13	1,870	15.85	11,797
eventbus	8,369	20	507	4	1.33	0	0.00	195	65.00	59	19.67	23	7.67	1	0.33	18	6.00	300
fastjson	179,996	158	3,863	8,205	49.88	77	0.47	4,255	25.87	1,264	7.68	243	1.48	387	2.35	2,019	12.27	16,450
glide	76,418	129	2,583	105	2.77	22	0.58	2,442	64.47	629	16.61	194	5.12	45	1.19	351	9.27	3,788
guice	72,980	59	1,931	178	2.85	46	0.74	3,871	61.92	1,043	16.68	216	3.45	51	0.82	847	13.55	6,252
hdiv	30,631	11	1,086	106	9.72	11	1.01	573	52.52	63	5.77	177	16.22	31	2.84	130	11.92	1,091
ical4j	24,130	35	2,303	132	11.22	15	1.28	682	57.99	167	14.20	48	4.08	2	0.17	130	11.05	1,176
j2objc	1,810,274	75	5,284	5,523	10.13	866	1.59	9,302	17.06	4,750	8.71	1,276	2.34	3,978	7.30	28,827	52.87	54,522
jenkins	175,150	654	31,156	658	6.15	161	1.51	3,273	30.61	794	7.43	314	2.94	185	1.73	5,308	49.64	10,693
jtk	204,105	9	1,373	2,627	13.03	4,557	22.60	1,008	5.00	55	0.27	37	0.18	1,068	5.30	10,813	53.62	20,165
junit4	31,242	151	2,474	55	3.15	18	1.03	985	56.38	248	14.20	32	1.83	47	2.69	362	20.72	1,747
keywhiz	23,337	32	1,538	89	5.67	23	1.46	1,036	65.99	178	11.34	90	5.73	14	0.89	140	8.92	1,570
libgdx	272,510	505	14,661	49,315	47.83	21,653	21.00	11,800	11.44	1,831	1.78	2,041	1.98	2,252	2.18	14,215	13.79	103,107
litiengine	75,877	20	3,324	316	11.86	46	1.73	771	28.94	448	16.82	253	9.50	21	0.79	809	30.37	2,664
lottie-android	16,258	102	1,292	80	7.41	104	9.64	442	40.96	145	13.44	126	11.68	21	1.95	161	14.92	1,079
mockito	55,751	220	5,523	234	9.87	12	0.51	1,288	54.35	285	12.03	126	5.32	38	1.60	387	16.33	2,370
mpandroidchart	25,232	69	2,068	134	6.85	36	1.84	385	19.69	232	11.87	155	7.93	38	1.94	975	49.87	1,955
nutch	141,710	43	3,215	236	7.68	28	0.91	1,353	44.01	467	15.19	113	3.68	164	5.34	713	23.19	3,074
okhttp	48,465	235	4,848	455	16.01	39	1.37	1,902	66.92	161	5.67	126	4.43	21	0.74	138	4.86	2,842
orienteer	55,681	12	2,274	63	2.68	27	1.15	1,122	47.77	584	24.86	395	16.82	22	0.94	136	5.79	2,349
picasso	9,136	97	1,368	64	8.82	36	4.96	546	75.21	27	3.72	10	1.38	7	0.96	36	4.96	726
rest-assured	73,511	105	2,020	121	5.85	32	1.55	1,440	69.57	288	13.91	107	5.17	14	0.68	68	3.29	2,070
rest.li	523,972	89	2,617	2,158	9.26	533	2.29	10,054	43.16	4,712	20.23	3,458	14.84	237	1.02	2,143	9.20	23,295
retrofit	26,513	152	1,865	60	2.49	7	0.29	1,691	70.14	352	14.60	18	0.75	6	0.25	277	11.49	2,411
riptide	27,072	18	2,131	4	0.52	0	0.00	650	85.08	22	2.88	46	6.02	8	1.05	34	4.45	764
rxjava	468,957	277	5,877	2,371	10.25	34	0.15	4,275	18.48	573	2.48	115	0.50	373	1.61	15,387	66.53	23,128
spring-boot	343,138	804	32,096	443	2.74	95	0.59	10,868	67.24	1,354	8.38	3,002	18.57	91	0.56	309	1.91	16,162
tomcat	343,703	61	23,140	1,142	6.68	263	1.54	7,374	43.16	1,675	9.80	696	4.07	846	4.95	5,089	29.79	17,085
twelvemonkeys	99,418	42	1,334	379	8.43	123	2.73	912	20.28	808	17.96	588	13.07	327	7.27	1,361	30.26	4,498
unirest-java	15,979	43	1,603	12	1.75	1	0.15	310	45.19	58	8.45	23	3.35	22	3.21	260	37.90	686
webmagic	12,926	40	1,119	28	2.87	3	0.31	763	78.26	80	8.21	27	2.77	10	1.03	64	6.56	975
xchart	24,406	50	1,451	119	7.93	31	2.07	628	41.84	338	22.52	50	3.33	26	1.73	309	20.59	1,501
zxing	107,064	109	3,582	208	9.78	137	6.44	695	32.68	267	12.55	108	5.08	157	7.38	555	26.09	2,127
Total	7,111,470	5,519	217,869	87,297	20.79	32,153	7.65	110,198	26.24	31,508	7.50	18,958	4.51	14,088	3.35	125,688	29.93	419,890

Table 2: C++ programs used in our experiment.

Project	LoC	Contributors	Commits	Kings		Median		Ditto		Cognome		Diminutive		Shorten		Index		Total
				Total	%	Total.	%	Total.	%	Total.	%	Total.	%	Total.	%	Total.	%	
asio	196,656	53	3,034	135	3.65	27	0.73	1,664	44.99	32	0.87	657	17.76	220	5.95	964	26.06	3699
assimp	614,926	462	10,934	78	6.76	74	6.41	739	64.04	10	0.87	94	8.15	13	1.13	146	12.65	1,154
bitcoin	541,474	853	32,661	46	4.58	27	2.69	621	61.79	8	0.80	11	1.09	39	3.88	253	25.17	1,005
bluematter	812,822	2	5	3,972	29.20	1,350	9.92	1,893	13.91	1,560	11.47	506	3.72	685	5.03	3,639	26.75	13,605
calligra	1,602,456	263	101,573	47	3.41	2	0.15	743	53.92	137	9.94	267	19.38	14	1.02	168	12.19	1,378
chaste	587,473	25	5,384	2,954	40.46	882	12.08	673	9.22	667	9.14	470	6.44	14	0.19	1,641	22.48	7,301
citra	428,966	222	9,141	27	5.11	19	3.60	255	48.30	4	0.76	36	6.82	27	5.11	160	30.30	528
clickhouse	1,422,903	921	83,445	114	4.13	40	1.45	2,228	80.78	66	2.39	108	3.92	14	0.51	188	6.82	2,758
core	9,262,610	25	3,058	4,044	5.29	1,516	1.98	45,465	59.47	10,741	14.05	10,799	14.13	420	0.55	3,459	4.52	76,444
freecad	4,842,675	383	27,647	528	6.94	210	2.76	4,705	61.83	100	1.31	513	6.74	181	2.38	1,372	18.03	7,609
gacui	504,062	3	2,238	8	0.62	50	3.91	576	45.00	44	3.44	294	22.97	15	1.17	293	22.89	1,280
gecko-dev	28,303,180	4,910	785,724	1,116	4.57	1,548	6.34	11,737	48.11	2,567	10.52	4,805	19.69	311	1.27	2,314	9.48	24,398
godot	4,976,013	1,590	41,538	525	9.87	270	5.08	1,711	32.17	128	2.41	1,934	36.36	107	2.01	644	12.11	5,319
gromacs	1,680,900	74	20,825	89	5.03	104	5.88	994	56.16	38	2.15	250	14.12	54	3.05	241	13.62	1,770
grpc	717,441	708	50,493	76	3.40	49	2.19	799	35.75	68	3.04	842	37.67	44	1.97	357	15.97	2,235
kdenlive	205,469	94	15,645	4	0.43	0	0.00	671	72.93	66	7.17	36	3.91	34	3.70	109	11.85	920
kdevelop	338,648	245	42,650	52	4.70	3	0.27	723	65.37	61	5.52	93	8.41	10	0.90	164	14.83	1,106
krita	983,754	336	57,706	80	5.93	12	0.89	573	42.48	109	8.08	216	16.01	44	3.26	315	23.35	1,349
lammps	1,626,808	185	29,307	281	11.35	56	2.26	1,272	51.37	199	8.04	169	6.83	85	3.43	414	16.72	2,476
mediapipe	235,825	2	111	11	1.54	47	6.58	511	71.57	13	1.82	1	0.14	26	3.64	105	14.71	714
mlir	75,845	2,285	415,644	9	5.70	18	11.39	83	52.53	24	15.19	8	5.06	2	1.27	14	8.86	158
mongo	5,015,374	571	63,227	917	3.17	381	1.32	14,644	50.66	761	2.63	2,770	9.58	2,019	6.99	7,412	25.64	28,904
mysql-server	3,733,193	88	170,220	803	6.94	124	1.07	7,941	68.60	713	6.16	949	8.20	141	1.22	904	7.81	11,575
obs-studio	482,886	477	10,466	22	3.42	9	1.40	429	66.72	57	8.86	59	9.18	5	0.78	62	9.64	643
opencv	2,166,493	1,360	31,603	1,598	11.96	859	6.43	5,672	42.45	367	2.75	376	2.81	730	5.46	3,761	28.14	13,363
openoffice	6,894,647	21	7,657	3,977	5.82	1,703	2.49	39,683	58.06	9,796	14.33	9,453	13.83	335	0.49	3,397	4.97	68,344
percona-server	3,777,210	238	185,334	849	7.35	127	1.10	7,887	68.32	712	6.17	913	7.91	142	1.23	914	7.92	11,544
proxysql	121,989	90	4,680	7	1.38	12	2.37	219	43.20	10	1.97	46	9.07	37	7.30	176	34.71	507
pytorch	1,792,819	2,155	43,944	56	2.10	111	4.15	1,472	55.07	35	1.31	164	6.14	115	4.30	720	26.94	2,673
qtbase	2,714,097	783	55,238	185	4.51	89	2.17	2,403	58.54	258	6.29	229	5.58	132	3.22	809	19.71	4,105
rocksdb	497,140	628	10,766	41	1.66	52	2.10	1,494	60.36	21	0.85	34	1.37	59	2.38	774	31.27	2,475
server	1,967,124	300	195,145	22	1.59	2	0.14	874	63.01	40	2.88	172	12.40	33	2.38	244	17.59	1,387
tensorflow	3,284,592	3,068	125,560	778	5.67	747	5.45	8,108	59.13	235	1.71	279	2.03	499	3.64	3,067	22.37	13,713
terminal	360,717	313	2,855	159	3.69	49	1.14	2,640	61.20	118	2.74	311	7.21	124	2.87	913	21.16	4,314
vtk	3,690,369	352	81,218	500	7.78	216	3.36	2,167	33.74	147	2.29	1,137	17.70	503	7.83	1,753	27.29	6,423
winget-cli	305,116	317	539	64	2.56	62	2.48	1,252	50.00	65	2.60	111	4.43	312	12.46	638	25.48	2,504
xbmc	1,094,954	785	59,641	42	9.77	2	0.47	208	48.37	29	6.74	83	19.30	20	4.65	46	10.70	430
yarp	1,029,531	77	17,416	45	2.25	18	0.90	1,021	51.13	91	4.56	352	17.63	65	3.25	405	20.28	1,997
yuzu	488,099	203	20,860	30	19.61	7	4.58	76	49.67	0	0.00	6	3.92	3	1.96	31	20.26	153
zerotierone	137,784	58	5,409	34	2.05	64	3.85	975	58.70	12	0.72	62	3.73	56	3.37	458	27.57	1,661
Total	99,515,040	25,525	2,830,541	24,325	7.28	10,938	3.27	177,801	53.24	30,109	9.01	39,615	11.86	7,689	2.30	43,444	13.01	333,921

4 Analyzing Identifier Names in CI/CD Context

This chapter will present what is a tool called GitHub Actions, that is used within CI/CD pipelines, and the development of our Action. This Action will apply our naming categories mentioned earlier to aid developers to name identifiers.

4.1 GitHub Actions

CI/CD practices can avoid mistakes and bugs during the development process (FOWLER; FOEMMEL, 2006), making small and consistent commits in short periods of time, it is possible to apply several tools in those commits. These tools in the GitHub environment are called GitHub actions. Using GitHub Actions makes it possible to automatize the whole CI/CD process, with deployment, testing, compiling, and code linter, among others. [1]. A single action is a code that runs in a specific GitHub Actions CI/CD workflow, like a testing or linting tool. Since this project aims to study properties of identifier naming, and hence the correlation between those names and code quality (LAWRIE, 2007) (BUTLER, 2010) (DEISSENBOECK; PIZKA, 2006), we proposed to create a static analysis tool that could enable the developer to visualize the current state of the names of identifiers at a certain period of time. This tool is used as a base for the eight naming categories presented earlier. A potential downside follows each naming category in code quality, and our tool can be used in the early steps of a CI/CD pipeline before the application's build and/or deployment. Thus, identifier names that follow under any category can be upgraded before creating pull requests or soliciting code reviews from peers.

Poor code quality and the presence of bugs are a downside of poor identifier naming (BUTLER, 2010). Therefore, these troubles could also be prevented using CI/CD practices and static analysis within it. By every push made by a developer, our tool scans the identifier names from the project and shows the developer the current state of the application in terms of the aforementioned naming categories. Naming impacts the code, and every naming category significantly impacts the overall code quality. Hence, the importance of discovering the prominence of naming categories in early development stages; the names that end up being labeled in a category that represents risks in code comprehension and quality can be improved in a consistent manner, always keeping up with the advances of the code. We argue that our tool, applying correct practices of CI/CD, can have a positive impact on the code's quality by shedding light on potential problems re-

[1] https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

garding comprehension, unsearchable code, misinformation, and difficulty in pronouncing that can occur when poor identifier naming is found.

4.2 Name Analyzer Action

The Identifier Analyzer Action was developed to increase the level of perception regarding identifier names in a Java/C++ project and leverages the GitHub actions environment to make it easier to locate identifiers that were named poorly in the early stages, avoiding potential pitfalls in code comprehension. It is possible to visualize names that can produce problems using our static analysis tool. The tool checks after every push on the repository, and then the developer is consistently reminded to look for their identifier names and improve them. Our tool follows the same logical sequence as our study; in short, first, we convert the source code to a like' representation, identify and separate the identifier names, and then, regarding our naming category, each name is assigned to a specific category based on its intrinsic syntax properties.

In order to be able to call the Id Analyzer Action in a GitHub Actions workflow, first we had to create a YAML (YAML ain't markup language) file that describes how our action image is going to run, which inputs are needed in order to run, and which are the expecting outputs, and a dockerfile pointing to that file. GitHub allows creating actions in two different ways: Create an action using Node.js and a Docker container. We created an Action using a Docker container, creating a reliable and consistent environment because the user does not have to worry about external tools or dependencies. Also, creating an action with a Docker container allowed us to use any programming language to create the action, not just JavaScript. Using Docker, when the action finishes running the workflow, GitHub creates an image based on our dockerfile file, and this file points to a ShellScript file. This ShellScript is responsible for all the processes necessary to run our action, from downloading dependencies, interacting with the file system, and running the process. This ShellScript is necessarily called "entrypoint.sh".

The first thing our entrypoint file does is download SrcML and convert our target Java/C++ code into XML files. This will create another folder that contains an XML file representing all source code, including specific markup tags that carry our desired identifier names. The next step is also downloading two Python dependencies, Pandas, which are responsible for analyzing data. After the installation is complete, three Python files are executed; the first performs several XPath Queries to recover identifier names and write them in a CSV file, alongside its type and scope. The following two Python files perform our tool's true objective, which is checking the names. One file manages the naming categories, checking the syntax of each name and determining if the name can be assigned to a category.

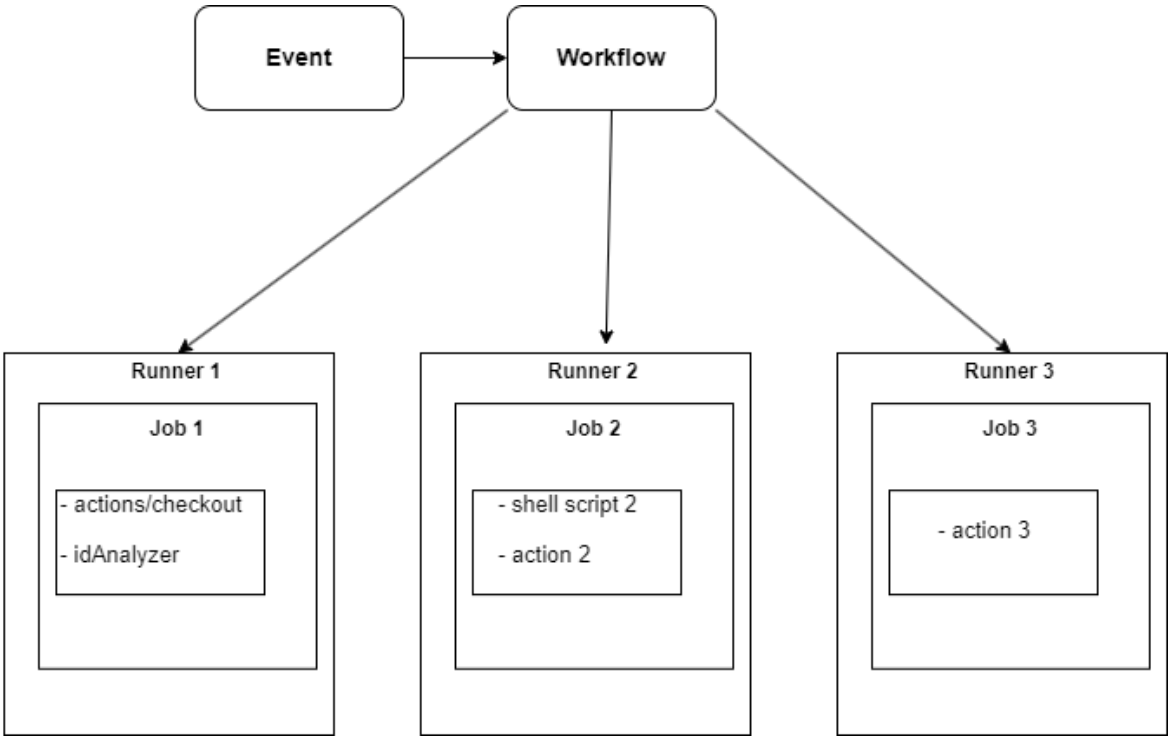
After the script checks the names, looking for correlations to address them to a category, the tool finishes its job by displaying the results on the screen. The results are presented by a list of names that belong to a category. Each category is labeled, and it is followed by all names in the project that correspond to that specific category, as well as the name of the file where the names can be found. The filename alongside each identifier name is crucial for the developer to locate the names more straightforwardly and to encourage the developer to look up the name and adapt it to each circumstance.

4.2.1 Tool in Action

It was established through the review that being careful with identifier naming is relevant for code comprehension and quality (BUTLER, 2010) (DEISSENBOECK; PIZKA, 2006) (LAWRIE, 2007), and that static analysis tools are an efficient manner to discover possible flaws (PRÄHOFER, 2012). Therefore, visualizing where the bad naming is happening is a way of shedding light on the problem, making it easier to fix it. With the aid of CI/CD practices, making short, concise, and smaller commits followed by a testing pipeline (FOWLER; FOEMMEL, 2006), we can also analyze naming in the pipeline. The usage of this tool, in a CI/CD context, informs the developer in each push, commit, or pull request, the names that can be harmful to its quality, allowing the developer to act quickly and change the names to more suited ones, avoiding naming an identifier with a name in any naming category mentioned in this study. In the result, we can see the name of each category, followed by a line consisting of the identifier name and the file it was declared.

In Figure 6 we can observe how a GitHub Action is triggered and the consequences of the Action. First of all, the developer can set an Event to trigger a workflow; an Event can be a pull request, the creation of an issue, a commit, etc. After the Event is triggered, the workflow is defined in *.github/workflows* YAML file is executed. Each workflow can contain one or more different jobs; a job is a set of steps in a workflow that are executed together inside a runner, the server from Github that actually runs the workflows. A step can either be an Action or a shell script. Our tool is an Action that can be triggered by an Event that calls the workflow to run our tool. In order to behave properly, before setting the step that calls our Name Analyzer Action, the developer needs to first call the *actions/checkout* action, an Action from GitHub that enables the runner to access the files from the repository. To check the result of our Action, the developer needs to enter the repository from his project and check the Actions section, which displays the current triggered Workflows and, within it, all jobs and steps that occur, together with the result from our Action.

Figure 6: View of a GitHub Workflow



In (DEISSENBOECK; PIZKA, 2006), it is said that there is a trend in corporate and proprietary code that they tend to have more specific words regarding their business than open code ones. Hence, this tool can be reproduced and advanced to capture their specificity regarding naming. Using this tool as an action of a open code repository, the predefined naming categories already capture different bad naming practices and explain the problems their use may occur from, but the final result can be expanded and replicated to each different scenario by designing a undesired naming pattern, and using the tool in CI/CD pipeline to check whether any identifier names ends up in the created category, shedding light to the problem, possibly avoiding bigger ones.

5 Conclusion

Coming up with proper identifier names is challenging (BROOKS, 1983). As stated by (HOST; OSTVOLD, 2007), even though programmers have to name identifiers on a daily basis, it still entails a great deal of time and thought. To make matters more challenging, identifier names are pivotal for program comprehension: developers have to go over identifier names to comprehend the code that they need to update, and poorly chosen names might hinder source code comprehension (AVIDAN; FEITELSON, 2017). Given that it has been estimated that identifiers contribute to about 70% of a software system's codebase (DEISSENBOECK; PIZKA, 2006), it cannot be disputed that there is a need to define what makes up a good identifier as well as to assist developers in naming identifiers. Similarly, identifying practices that result in poor identifier names might enhance programmers' awareness and contribute to improving educational materials and code review methods. As an initial foray into creating an approach to optimal identifier naming (i.e., how to assign the proper words to an identifier), we investigated eight naming practices categories "in the wild". The categories provide examples of naming practices from real-world software projects. We illustrated their possible consequences and also outlined their prevalence across projects and code contexts (i.e., ATTRIBUTE, PARAMETER, METHOD, FOR, WHILE, IF, and SWITCH).

Our results, based on 2,603,381 identifier names extracted from 80 real-world Java and C++ projects and on a survey, would seem to suggest the following:

- The eight categories are recurrently found in practice, but two are more common in Java and C++ projects: naming identifiers with the same name as her *Type* (*Ditto*) and use single-letter names denoting counters (*Index*). Specifically, *Index* and *Ditto* are by far the most frequently occurring naming practices across Java projects: *Index* occurrences account for approximately 30% of all naming practice occurrences in the examined Java projects, while *Ditto* occurrences amounted to roughly 27%. As for C++ programs, *Ditto* is the most widely used naming practice, which accounts for around 54% of all naming practice occurrences. *Index* and *Diminutive* are also popular among C++ coders, accounting for 13% and 11% of all naming practice occurrences. *Shorten* seems to be the least used naming practice both by Java and C++ programmers. Additionally, programmers seem to be hardly influenced by IDE-like features that help them to choose identifier names, although only 12 out of 52 surveyed programmers (23%) acknowledged a tendency to write identifier names spelled in the same way as their *Types*;

- There are several very common names (e.g., `value`; `result`; and `name`) and recurrent single-letter names (e.g., `i`, `e`, `s`, `c`) used in practice. The lion's share of these names are used to denote identifiers that store either `integer` or `string` values. According to our results, single-letter identifiers are more commonly used by Java programmers: `i`, `e`, `s`, `c`, `t`, `a`, `b`, `p`, and `n` would seem to be widely used by programmers. In C++ (in contrast to Java), coders tend to prefer a smaller set of single-letter names: `i`, `e`, `s`, `s`, `c`, `t`, `a`, `b`, `p`, and `n`. Thus, differently from Java, in C++ `e`, `c`, `t`, and `a` do not rank among the most common single-letter identifier names;

- The programmer's naming practices are context-specific: single-letters names (*Index* and *Shorten*) seem to be more common in short-scope contexts (IF, FOR, WHILE), although they can also be found in large-scope contexts (e.g., ATTRIBUTE). Results from our survey questionnaire showed that programmers acknowledge that the *Index* naming practice occurs mainly inside contexts surrounded;

- *Diminutive* is the most adopted naming category practice by survey respondents and *Median* is the least used naming practice. All the respondents adopt at least one naming practice category "Occasionally" or "Often".

- We could benefit from including poor naming practices in code reviews. The current practices follow extensive checklists, but no one addresses naming issues. A more nuanced take is to consider variable names that depart from commonly used naming practices as elements that can lead to a source of problems.

We also designed a tool that reads all identifier names in a project and returns the prevalence of names that belong to any naming category. We argue that with the correct application of CI/CD practices (FOWLER; FOEMMEL, 2006), our tool can benefit developers by reassuring the current state of the identifiers based on naming categories. Each category can reproduce bad practices that damage code quality, and then showing the developer each name that belongs to a category is an effective way to encourage developers to change the possible harmful names to names that don't follow any naming category practice. Our tool serves as a constant reminder that code evolves, and names should also evolve alongside it, and the one responsible for that task is the developer writing the code.

We believe our results have the potential to inspire several future research directions. Our work highlights the need for further research on how naming practices are prevalent in source code and how better names can be chosen. In this direction, an aspiring goal would be to devise tools capable of automatically evaluating and suggesting renaming opportunities during code review. Similarly, code generation tools can capitalize on commonly used naming practices to generate names automatically. Additionally, since our results would seem to suggest that some identifier names are context-dependent,

we believe that tools (e.g., IDE-based identifier name recommendation system) can take advantage of context information during software development by constantly monitoring how programmers name identifiers so that they can help developers new to a given project through the automated recognition of context- and project-specific naming conventions. Therefore, this automated identifier naming assistant can support developers by identifying inappropriate naming choices and making recommendations. As a result, our long-term goal is to support the identification of opportunities to rename identifiers and understand more about programmers' naming practices. Finally, in future work, we plan to perform a qualitative study on commits, code changes, and review discussions. Another possible future research avenue would be to account for the role of human factors in choosing identifier names by exploring how programmer experience, team size, and mood influence naming practices throughout different software projects. Regarding our Name Analyzer tool, we plan to include new features in future work to enhance the quality of the results. Artificial intelligence can be used to check the names belonging to a category and, based on the whole project, suggest proper names to substitute for the original ones.

Although our results give practitioners and researchers alike a good glimpse into the most common options for naming identifiers in C++ and Java, we did not investigate how each naming practice contributes, if at all, to improving code comprehension. Therefore, future research efforts should aim to better understand how these commonly used naming practices influence readability during code comprehension.

5.1 Threats to Validity

As with most empirical studies, our study also has some practical limitations, i.e., it is also subject to some threats to its validity. In this section, we present potential threats and how we tried to mitigate some of those issues. Conclusion and External Validity One potential threat is that the samples we used in our study might not be representative of the target population: our analysis took into account 40 open-source Java projects and 40 C++ projects. To mitigate this threat concerning the conclusion and generalization of the study results, we tried to select a heterogeneous sample. We think the impact of this threat is minimal for three reasons: (i) Java and C++ are two popular programming languages;8 (ii) our sample covers somewhat small code-bases (with less than 10K LoC) and large-scale ones (with over 100K LoC), and (iii) we selected projects from a broad range of domains. Thus, we argue that our study can be seen as an initial step towards identifying trends Java and C++ programmers follow when picking identifier names. However, given the sizes of our samples, we cannot rule out the possibility that our results do not reflect how Java and C++ programmers name identifiers. That is, the results might not be generalizable beyond the study samples and the participants that took part in our survey. To understand the prevalence of naming categories across

Java and C++ projects, we employed a set of metrics: program size (LoC), number of commits, and number of contributors. Nevertheless, as with many software metrics, one potential threat is that these measurements might not be sophisticated enough for our investigation. Thus, our findings might not carry over to other settings and similar programming languages. It is also worth emphasizing that context and scope would seem to play an important role in determining identifier names. For instance, some of the most common identifier names listed in Table 3 would seem to be context-dependent, e.g., node. We surmise that is the case because programmers might want to include relevant domain information when turning concepts into names. Although we tried our best to maximize the sample heterogeneity during sample selection, we cannot rule out the fact that the most common domains (e.g., XML file parsing) from which the programs in our sample were extracted might have an impact on variable naming. Finally, the representativeness of the survey respondents cannot be guaranteed. Our target population was programmers, but we did not take any measures to verify the identity of the respondents. However, we have included two initial questions, which might have permitted us to filter out individuals not belonging to our target population. There might also exist some other factors that bias our conclusions. One example is the environment in which the respondents worked. Another one is whether or not respondents have a correct understanding of each category. To mitigate the latter, we included in the questionnaire a brief description and an example of the categories. Future studies can ask respondents to consider this factor and evaluate how it impacts the naming practice category adoption.

5.2 Construct & Internal Validity

A threat to the construct validity of our study comes from the number of identifier names we analyzed in our study. It might be argued that a more significant amount of names may lead to better and more conclusive results. To mitigate this threat we analyzed 2,603,381 identifier names in highly diverse sets of Java and C++ projects. Additionally, another potential threat has to do with how well the naming practices we identified reflect extant research and current industry practices. We tried to mitigate this threat by drawing from previous research, which helped us to get a better understanding regarding whether or not some of the naming practices we identified are indeed recurring practices. We also conducted a survey with 52 participants in order to gather programmers' perceptions about the use and occurrence of the investigated naming practices. We tried to minimize possible construct and internal validity associated with the survey by disseminating it online through multiple websites and online groups; and introducing a brief description and an example of each question.

Bibliography

ALLAMANIS, M.; BARR, E. T.; BIRD, C.; SUTTON, C. Learning natural coding conventions. In: *International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014. Citado 2 vezes nas páginas 10 and 33.

BUTLER, S.; WERMELINGER, M.; YU, Y.; SHARP, H. Exploring the influence of identifier names on code quality: An empirical study. In: IEEE. *2010 14th European Conference on Software Maintenance and Reengineering*. [S.l.], 2010. p. 156–165. Citado 7 vezes nas páginas 10, 11, 12, 13, 15, 41, and 43.

AVIDAN, E.; FEITELSON, D. G. Effects of variable names on comprehension: An empirical study. In: *25th International Conference on Program Comprehension*. [S.l.: s.n.], 2017. Citado 7 vezes nas páginas 10, 13, 14, 29, 32, 34, and 45.

HOFMEISTER, J.; SIEGMUND, J.; HOLT, D. V. Shorter identifier names take longer to comprehend. In: IEEE. *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*. [S.l.], 2017. p. 217–227. Citado 5 vezes nas páginas 10, 11, 15, 24, and 29.

LAWRIE, D.; MORRELL, C.; FEILD, H. Effective identifier names for comprehension and memory. *Innovations Syst Softw Eng*, Springer, v. 3, n. 1, p. 303—-318, 2007. Citado 3 vezes nas páginas 10, 15, and 43.

FAKHOURY, S.; MA, Y.; ARNAOUDOVA, V.; ADESOPE, O. The effect of poor source code lexicon and readability on developers' cognitive load. In: *International Conference on Program Comprehension*. [S.l.: s.n.], 2018. Citado na página 10.

DEISSENBOECK, F.; PIZKA, M. Concise and consistent naming. *Software Quality Journal*, Springer, v. 14, n. 3, p. 261–282, 2006. Citado 10 vezes nas páginas 10, 11, 13, 14, 24, 35, 41, 43, 44, and 45.

HOST, E. W.; OSTVOLD, B. M. The programmer's lexicon, volume i: The verbs. In: *International Working Conference on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2007. Citado 2 vezes nas páginas 10 and 45.

MARTIN, R. C. Clean code: A handbook of agile software craftsmanship.(2008). *Citado na*, p. 19, 2008. Citado 6 vezes nas páginas 10, 14, 24, 25, 26, and 27.

MARCUS, A.; SERGEYEV, A.; RAJLICH, V.; MALETIC, J. I. An information retrieval approach to concept location in source code. In: IEEE. *11th working conference on reverse engineering*. [S.l.], 2004. p. 214–223. Citado na página 10.

WAINAKH, Y.; RAUF, M.; PRADEL, M. Idbench: Evaluating semantic representations of identifier names in source code. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2021. Citado na página 10.

SCHANKIN, A.; BERGER, A.; HOLT, D. V.; HOFMEISTER, J. C.; RIEDEL, T.; BEIGL, M. Descriptive compound identifier names improve source code comprehension. In: *International Conference on Program Comprehension*. [S.l.: s.n.], 2018. Citado na página 10.

ARNAOUDOVA, V.; PENTA, M. D.; ANTONIOL, G. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, Springer, v. 21, n. 1, p. 104–158, 2016. Citado 3 vezes nas páginas 10, 13, and 24.

GRESTA, R.; CIRILO, E. Contextual similarity among identifier names: An empirical study. In: SBC. *Anais do VIII Workshop de Visualização, Evolução e Manutenção de Software*. [S.l.], 2020. p. 49–56. Citado na página 11.

GRESTA, R.; CIRILO, E. Say my name! an empirical study on the pronounceability of identifier names. In: SBC. *Anais do IX Workshop de Visualização, Evolução e Manutenção de Software*. [S.l.], 2021. p. 51–55. Citado na página 11.

CHARITSIS, C.; PIECH, C.; MITCHELL, J. Assessing function names and quantifying the relationship between identifiers and their functionality to improve them. In: *Conference on Learning@ Scale*. [S.l.: s.n.], 2021. Citado na página 11.

NYAMAWE, A. S.; BAKHTI, K.; SANDIWARNO, S. Identifying rename refactoring opportunities based on feature requests. *International Journal of Computers and Applications*, Taylor & Francis, p. 1–9, 2021. Citado na página 11.

LAWRIE, D.; MORRELL, C.; FEILD, H.; BINKLEY, D. What's in a name? a study of identifiers. In: IEEE. *14th IEEE International Conference on Program Comprehension (ICPC'06)*. [S.l.], 2006. p. 3–12. Citado na página 13.

TOFTE, M.; TALPIN, J.-P. Region-based memory management. *Information and computation*, v. 132, n. 2, p. 109–176, 1997. Citado na página 13.

TAKANG, A. A.; GRUBB, P. A.; MACREDIE, R. D. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, v. 4, n. 3, p. 143–167, 1996. Citado 2 vezes nas páginas 13 and 15.

LI, G.; LIU, H.; LIU, Q.; WU, Y. Lexical similarity between argument and parameter names: An empirical study. *IEEE Access*, IEEE, v. 6, p. 58461–58481, 2018. Citado na página 13.

KAWAMOTO, K.; MIZUNO, O. Predicting fault-prone modules using the length of identifiers. In: IEEE. *2012 Fourth International Workshop on Empirical Software Engineering in Practice*. [S.l.], 2012. p. 30–34. Citado 2 vezes nas páginas 13 and 15.

FEITELSON, D.; MIZRAHI, A.; NOY, N.; SHABAT, A. B.; ELIYAHU, O.; SHEFFER, R. How developers choose names. *IEEE Transactions on Software Engineering*, IEEE, 2020. Citado 2 vezes nas páginas 14 and 32.

SANTOS, R. M. dos; GEROSA, M. A. Impacts of coding practices on readability. In: *Internation Conference on Program Comprehension*. [S.l.: s.n.], 2018. Citado na página 14.

CAPRILE, C.; TONELLA, P. Nomen est omen: Analyzing the language of function identifiers. In: IEEE. *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. [S.l.], 1999. p. 112–122. Citado na página 14.

GRESTA, R.; CIRILO, E. Contextual similarity among identifier names: An empirical study. In: SBC. *Workshop de Visualização, Evolução e Manutenção de Software*. [S.l.], 2020. p. 49–56. Citado na página 15.

SKA, Y.; SYED, H. H. A study and analysis of continuous delivery, continuous integration in software development environment. *SSRN Electronic Journal*, v. 6, n. 09, 2019. Citado na página 15.

SHAHIN, M.; BABAR, M. A.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, v. 5, p. 3909–3943, 2017. Citado 3 vezes nas páginas 15, 16, and 17.

RANGNAU, T.; BUIJTENEN, R. v.; FRANSEN, F.; TURKMEN, F. Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In: IEEE. *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. [S.l.], 2020. p. 145–154. Citado na página 16.

LOURIDAS, P. Static code analysis. *Ieee Software*, IEEE, v. 23, n. 4, p. 58–61, 2006. Citado na página 16.

PRÄHOFER, H.; ANGERER, F.; RAMLER, R.; LACHEINER, H.; GRILLENBERGER, F. Opportunities and challenges of static code analysis of iec 61131-3 programs. In: IEEE. *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. [S.l.], 2012. p. 1–8. Citado 2 vezes nas páginas 16 and 43.

ZHENG, J.; WILLIAMS, L.; NAGAPPAN, N.; SNIPES, W.; HUDEPOHL, J. P.; VOUK, M. A. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering*, IEEE, v. 32, n. 4, p. 240–253, 2006. Citado na página 16.

FOWLER, M.; FOEMMEL, M. *Continuous integration*. 2006. Citado 5 vezes nas páginas 16, 17, 41, 43, and 46.

SINGH, C.; GABA, N. S.; KAUR, M.; KAUR, B. Comparison of different ci/cd tools integrated with cloud platform. In: IEEE. *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. [S.l.], 2019. p. 7–12. Citado na página 17.

COLLARD, M. L.; DECKER, M. J.; MALETIC, J. I. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In: IEEE. *2013 IEEE International Conference on Software Maintenance*. [S.l.], 2013. p. 516–519. Citado na página 20.

BENIAMINI, G.; GINGICHASHVILI, S.; ORBACH, A. K.; FEITELSON, D. G. Meaningful identifier names: The case of single-letter variables. In: *International Conference on Program Comprehension*. [S.l.: s.n.], 2017. p. 45–54. Citado 8 vezes nas páginas 24, 26, 29, 32, 33, 34, 36, and 38.

ALSUHAIBANI, R. S.; NEWMAN, C. D.; DECKER, M. J.; COLLARD, M. L.; MALETIC, J. I. On the naming of methods: A survey of professional developers. In: *International Conference on Software Engineering*. [S.l.: s.n.], 2021. Citado 3 vezes nas páginas 24, 26, and 27.

DILEO, C. Clean ruby. *Citado na*, 2019. Citado 3 vezes nas páginas 24, 25, and 26.

BROWN, W. H.; MALVEAU, R. C.; MCCORMICK, H. W. S.; MOWBRAY, T. J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. ed. USA: John Wiley & Sons, Inc., 1998. ISBN 0471197130. Citado na página 24.

SWIDAN, A.; SEREBRENIK, A.; HERMANS, F. How do scratch programmers name variables and procedures? In: *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.: s.n.], 2017. p. 51–60. Citado na página 32.

KERNIGHAN, B. W.; PIKE, R. *The Practice of Programming*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1999. Citado na página 34.

SCALABRINO, S.; BAVOTA, G.; VENDOME, C.; LINARES-VÁSQUEZ, M.; POSHYVANYK, D.; OLIVETO, R. Automatically assessing code understandability: How far are we? In: *International Conference on Automated Software Engineering*. [S.l.: s.n.], 2017. Citado na página 35.

JIANG, L.; LIU, H.; JIANG, H. Machine learning based recommendation of method names: How far are we. In: *International Conference on Automated Software Engineering*. [S.l.: s.n.], 2019. Citado na página 36.

ISOBE, Y.; TAMADA, H. Are identifier renaming methods secure? In: *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. [S.l.: s.n.], 2018. Citado na página 36.

PERUMA, A.; MKAOUER, M. W.; DECKER, M. J.; NEWMAN, C. D. An empirical investigation of how and why developers rename identifiers. In: *2nd International Workshop on Refactoring*. [S.l.: s.n.], 2018. Citado na página 36.

PERUMA, A.; MKAOUER, M. W.; DECKER, M. J.; NEWMAN, C. D. Contextualizing rename decisions using refactorings and commit messages. In: *International Working Conference on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2019. Citado na página 36.

LAWRIE, D.; FEILD, H.; BINKLEY, D. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, Springer, v. 12, n. 4, p. 359–388, 2007. Citado na página 41.

BROOKS, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, v. 18, n. 6, p. 543–554, 1983. Citado na página 45.

.1 Survey Questionnaire

Education level

Experience in software development

1. How often do you choose identifier names with numbers at the end?
Examples: People people1; People people2

Where do you usually see identifier names with numbers at the end?

Attributes *Methods* *Loops* *Conditionals* *None*

2. How often do you choose identifier names with numbers in the middle?
Example: Char int2char

Where do you usually see identifier names with numbers in the middle??

Attributes *Methods* *Loops* *Conditionals* *None*

3. How often do you name identifiers after their *Type* names?
Examples: String string, People people

Where do you usually see identifier names spelled in the same way as their *Types*?

Attributes *Methods* *Loops* *Conditionals* *None*

4. How often do you name identifiers as chunk of their respective *Type* name?
Examples: EngineExecutionTestListener listener

Where do you usually see identifier names as chunk of their respective *Type* name?

Attributes *Methods* *Loops* *Conditionals* *None*

5. How often do you includes in identifier names an additional suffix or prefix that is the name of the respective *Type*?
Examples: String nameString

Where do you usually see identifier names containing an additional suffix or prefix that is the name of the respective *Type*?

Attributes *Methods* *Loops* *Conditionals* *None*

6. How often do you choose single-letter identifier names?
Examples: Integer j

Where do you usually see single-letter identifier names?

Attributes *Methods* *Loops* *Conditionals* *None*

7. How often do you name identifiers with the starting letters that correspond to their respective *Types*?
Examples: People p

Where do you usually see names which are the starting letters that correspond to their respective *Types*?

Attributes *Methods* *Loops* *Conditionals* *None*