University of Bergen
Department of Informatics

Development of a System for Analysing Method Names in Java Source Code

Author: Emily Mi Luong Nguyen
Supervisors: Mikhail Barash, Anya Helene Bagge

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

June 1, 2022

Abstract

Naming code can seem like a simple task, however finding a good name can be rather challenging. Entity names should be consistent and brief yet comprehensive when representing the information each entity hold. What is considered a good name can be highly debatable, although it usually involves descriptive names that can contribute to readability and comprehensibility of source code. Bad code names can cause uncertainty, potential future bugs and be misleading. For this reason, the task of naming code is vital, hence there is a need of a system to improve and maintain it. To develop such a system, there are requirements required to be specified to define the expected implementation for certain entity names. These requirements are encoded into software in a domain-specific language, granting executable code to be generated from the expressed requirements. As a result, this name analysis tool provides programmers to perform code analysis on Java source code checking if the entities act in accordance with the requirements of their names. Additionally, the result shows insights of how contributions from linguistics can be valuable for software development and can be used to analyse software languages, such as entity names.

Acknowledgements

First of all, I would like to express my deepest thanks to my supervisors, Mikhail Barash and Anya Helene Bagge for guiding and supporting me throughout this thesis. I am indebted to Mikhail for all the invaluable discussions, sharing of ideas, patience and regular follow up. Especially for helping me shape my thesis to fit my interests, making this journey fun, insightful and interesting for me. Thanks should also go to Anya for constructive criticism, and who inspired me to choose this field in software engineering.

I would like to extend my gratitude to Knut Anders Stokke for his generousity in assistance when I needed it. I would also like to express my deepest appreciation to my best friend Michael for always checking up on me, keeping in lane and making sure I always do my best.

Last but not least, I am extremely lucky to have a family who are considerate, loving, accepting and supportive of my journey to pursue this dream. I could not have undertaken this journey without the love from everyone around me.

Emily Mi Luong Nguyen
June 1, 2022

Contents

1 Introduction **1**

2 Background **3**

2.1 Java . . . 3

2.2 Names in Software Languages . . . 4

2.2.1 Importance of Names . . . 4

2.2.2 Difficulty in Naming . . . 7

2.2.3 Name Quality . . . 8

2.3 Approaches From Linguistics . . . 10

2.3.1 Natural Language Processing . . . 11

2.4 Domain-Specific Languages . . . 14

2.5 Eclipse Xtext . . . 16

2.6 JavaParser . . . 18

3 Tool for Analysing Names **24**

3.1 The Analysis Tool . . . 24

3.1.1 Requirements . . . 24

3.1.2 Interactive Command Line . . . 32

4 Implementation **35**

4.1 Architecture . . . 35

4.2 DSL and Java . . . 36

4.2.1 Grammar . . . 38

4.2.2 Code Generation . . . 38

4.2.3 Name Analyser . . . 38

5 Evaluation: The Programmer's Lexicon **43**

5.1 Case Study . . . 43

5.2 The Lexicon . . . 44

5.3 Retrospective . . . 49

6 Related Work **50**
6.1 Naming in Software Languages 50
6.2 Linguistics in Software Languages 51

7 Conclusion and Future Work **53**

Bibliography **57**

A Declarations and Rules Implementation **65**

B Lexicon With Implementation **71**

C Grammar for the Implemented DSL **94**

D Code Generator for the Implemented DSL **97**

List of Figures

2.1 Post-order tree traversal sample in Java. 6
2.2 Hyponyms and hypernyms example. 13
2.3 JavaParser library AST example. 19
2.4 JavaParser YAML printer implementation in the `printAST` Java class. . . 19
2.5 An example Java source code file called `GoodNumber`. 20
2.6 JavaParser YAML output of the `GoodNumber` Java class. 21
2.7 JavaParser AST representation of the `GoodNumber` Java class. 22
2.8 JavaParser library code example of printing all method names. 23

3.1 Requirement specifications example. 25
3.2 Interactive command line. 34
3.3 Interactive command line—choice: P. 34
3.4 Interactive command line—choice: I. 34

4.1 Overview of architecture. 37
4.2 Sample of DSL grammar in Xtext. Full grammar can be found in Appendix C. 39
4.3 Sample of the code generator in Xtend. Full code generator can be found in Appendix D. 40
4.4 The Java method `ruleChecker`. Full implementation of analysis can be found in the repository for Java project. 42

5.1 Implementation of `create` in our DSL according to definition in lexicon. 45
5.2 Implementation of `is` in our DSL according to definition in lexicon. . . . 46
5.3 Java source code example. 46
5.4 Modified implementation of `create` in our DSL. 47
5.5 Modified implementation of `is` in our DSL. 47
5.6 Interactive command line—analysis report. 48

List of Tables

2.1 Sample of naming convention from Butler et al. [14]. 9
2.2 Name mold example [24]. 10

3.1 Valid comparison operators. 27
3.2 Part of speech tags used in the Penn Treebank Project [54]. 30
3.3 Name condition and corresponding name examples. 32
3.4 Lexicon Terminology [39]. 33

Chapter 1

Introduction

The former executive officer at Microsoft, Bob Muglia, once said at a Visual Studio event in 2010 that "There is no question that the world runs on software. Just look around you. Look at everything." [59]. And indeed, software seems to be the underlying power of every system and technology that we interact daily with in our lives today. The world is steadily becoming more digitalised for every day that passes by [79].

Correspondingly, there is a growth of open-source software (OSS) [26]. Prominent examples of open-source software are the Apache Software Foundation [29] and the Eclipse Foundation [27]. In fact, Apache Software Foundation claim to be the biggest open-source foundation with over 49 000 code contributors [29]. Open-source software stimulate the development of open collaboration contributed by the volunteers, where contributors benefit from and build on each others skills [51]. In addition, this can strengthen the trust between the software and the users, due to the fact that the users can contribute to changes and/or improvements. As a result, a good amount of proprietary software depend on open-source software.

Software quality is a crucial substance for a successful software programme [42], whether it is of a proprietary or open-source software. Code readability and comprehension are two of many factors assuring software quality [20, 21]. Lawrie et al. demonstrate that names of entities—such as method and variable names—affect code comprehension, due to such names being one of two main sources of information about a domain [49]. Therefore, programmers follow naming conventions to maintain code quality [16]. However, challenges of manually following coding conventions throughout the whole code base may arise, especially in large sized code base. Likewise, although coding conventions are

followed, there is no guarantee that programmers will choose names that correctly and/or precisely represent the implementation of the entities.

These considerations illustrate an important aspect: the art of naming is vital in programming and there is a need of a system to improve it. This allows us to formulate the goal of this thesis, which can be represented by the following problem statement: *To explore how a domain-specific language (DSL) with approaches from linguistics can be designed to allow programmers to express a system for analysing names in Java source code.* This problem statement encompasses the following research questions: Are approaches from linguistics useful for Java name analysis? And if so, how can approaches from linguistics be used to analyse names in Java programs?

Thus far, this chapter introduces the motivation and research goal of the work described in this thesis. The remaining structure is as follows:

Chapter 2 presents an overview of the background. This includes a brief description of the programming language Java, an explanation of the roles that names in software languages can have and an introduction of linguistic approaches. Additionally, the last part of the background discusses the notion of domain-specific languages (DSLs), the language workbench Eclipse Xtext and the JavaParser library.

Chapter 3 presents the name analysis tool. This includes an in-depth description of the usage of the tool involving code examples.

Chapter 4 presents the implementation of the work in this thesis. It starts by explaining the architecture of all the involved components. Then, describes how the technical work is implemented in Java, Xtext and Xtend.

Chapter 5 presents an evaluation of the developed name analysis tool. The thesis conducts a case study and implements the lexicon in the case study. A description of a few implemented entries will be given.

Chapter 6 presents the work related to this thesis. This encompasses research around the act of naming in software languages, as well as how linguistics have been used to support software programs.

Chapter 7 presents a conclusion and suggestion of several potential directions for further development of the work in this thesis.

Chapter 2

Background

In this chapter, we present a brief overview of the background the thesis is based upon. We start with some details of the chosen programming language, Java. Following this, we continue with a discussion on the importance and significance of names in software languages, and then present some approaches from linguistics that we employ in the thesis. Additionally, we discuss the notion of a domain-specific language (DSL), and give an overview of the language workbench Eclipse Xtext, used to implement DSLs. Finally, we give a brief introduction to JavaParser, which is a Java open-source library for parsing Java source code.

2.1 Java

The Java programming language is among one of the most popular programming languages still actively in use today. It is a high-level, object-oriented, concurrent, strongly-typed and class based general-purpose programming language (GPL) developed by *Sun Microsystems* that *Oracle Corporation* later acquired [66, 67]. GPLs are languages for computer software, designed to build various of software and applications, without being limited to a specific domain or set of tasks. Java is known for being designed so that developers can *write once, run anywhere* (WORA).

According to the annual Developer Ecosystem Survey conducted by JetBrains[1], Java is the most used programming language in countries such as South Korea, China and

[1] `https://www.jetbrains.com`

Germany in 2021 [41]. It is especially a popular choice in desktop computing, mobile computing, numerical computing and games [34]. The following list shows a few usage activities of Java during the last decade [68].

- In 2012–2014, up to 97% of enterprise desktops ran Java.
- In 2015, 13 billion devices ran Java.
- In 2017, there were 21 billion cloud-connected Java Virtual Machines (JVMs).
- In 2020, Java remain on the top as number one programming language for developers

The architecture of Java includes three components that are fundamental: *Java SE Runtime Environment* (JRE), *Java SE Development Kit* (JDK) and *Java Virtual Machines* (JVM) [66]. The JRE is the environment needed to execute software written in Java. It provides various JVM, libraries and other necessary components as part of the JDK. The JDK is the development kit required to develop Java software. It includes the JRE and several development tools like compilers and debuggers. The JVM is a virtual machine that can be on several platforms making it possible to run Java software anywhere. The virtual machine component is the reason Java is considered WORA.

2.2 Names in Software Languages

Through good software quality assurance methods, a programme is more likely to be much more efficient and reliable. One of the dimensions that affect software quality is naming. Naming code entities can be among one of the hardest, yet important and major part of coding; thus, it is essential to invest in and focus on names in software languages. Karlton is famous for demonstrating this with the following saying: "There are only two hard things in Computer Science: cache invalidation and naming things."[2].

2.2.1 Importance of Names

Names of entities play a significant part in code quality [2]. A good name has the ability to help programmers gain a deeper understanding of the code more efficiently. Being able to analyse and modify code, one must first understand what information names of entities hold, like for example, the underlying information of a variable. Thinking about the code

[2] `https://martinfowler.com/bliki/TwoHardThings.html`

can be extremely challenging if it is not obvious what a certain variable represents. This is the reason for why good entity names can be quite useful to help readers write and comprehend code better.

There are several entities that programmers need to name. According to Hermans, such entities can be referred to as *identifiers* [36]. Identifiers can be *variables*, *methods*, *functions*, *modules*, *libraries*, *namespaces* or types like *classes*, *interfaces*, *structs*, *delegates*[3] or *enums*. Although, all of these entities are categorised as an identifier, they are named differently. For example, class names are different from method names, method names are different from variable names [18, 38]. Why names are so influential will be discussed in the following paragraphs.

Generally, names occur quite frequently in code, which makes up a great deal of a code base. Approximately more than 70% of all characters in the source code of Eclipse are identifiers [21]. This essentially means that code names are something that most programmers will be *reading*. Therefore, if the names are discursive, it will naturally be problematic. Names need to be *concise* and *consistent*.

Furthermore, the amount of times a programmer refers to names when reviewing code should be taken into account. Research indicate that programmers regularly talk of and rely on names to comprehend the behaviour of a programme during code review discussions or maintenance [87, 2]. Inspecting peers' source code manually is a practice that is still valuable for detecting software defects and adhere to team standards [1]. Allamanis et al. [2] examined in total 169 code reviews from randomly selected product groups from Microsoft. Among these code reviews, 18% were coding convention feedback, 9% talked about identifier names and 2% suggested coding formatting changes. This further shows that even after the code is completed, peers might not be satisfied with the name and suggest for changes or discussions.

Another reason for why names matter is for the fact that they can serve as a form of documentation [24]. According to Feitelson et al., not only are names implicitly documentations, but there are cases of names being the *only* documentation in the code [24]. In addition, names as documentation are accessible wherever they are written. As they make up a big deal of the code base, they become the most read documentation alongside with comments in the code [36].

[3] `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/`

Beacon

Names in software programs can be great *beacons* [12, 86]. A beacon is usually a piece of code, name or feature that indicates what type of structure or operator is present in the source code. This can ease code comprehension, due to the ability for programmers to see if a programme contains any particular algorithm, data structure, operators or other patterns more efficiently. To demonstrate the definition of a beacon, using an example from Hermans [36], one can say that in a programme that has variable names called *tree* and *root*, with the field names *left* and *right*, the peer reviewers can reason about this programme having a data structure of a binary tree. Figure 2.1 is a post-order tree traversal in Java that shows an example of this.

```
1  class Node {
2      int key;
3      Node left, right;
4      public Node(int item) {
5          key = item;
6          left = right = null;
7      }
8  }
9
10 class BinaryTree {
11     Node root;
12     BinaryTree() { root = null; }
13     void printPostOrder(Node node) {
14         if (node == null)
15             return;
16         printPostOrder(node.left);
17         printPostOrder(node.right);
18     }
19 }
```

Figure 2.1: Post-order tree traversal sample in Java.

As identifier names seem to be more important than just indicating the semantics of the identifiers, they can be quite useful when used right. Names are in the majority of code base, ergo they are crucial. When programmers choose poor names for their identifiers, for example, creating names that involve unrelated words to the implementation of the identifier or violate naming conventions, it can demolish all the features that a name can have. Not only can poor names have less functions, but they can also increase software defects with error insertions or be misleading [36]. Consequently, names are extra important; good names increase comprehension, bad names increase software bugs.

2.2.2 Difficulty in Naming

Naming identifiers is not as easy as it may seem to be [12]. Indeed, it is difficult to select a name that accurately represents a piece of source code yet at the same time is clearly understood and interpreted the same way by all readers. When reading a call to a method with a vague name, one will need to spend some time looking at the implementation in order to understand what the method does. There is currently no agreed standardised format among programmers on how to name code; these tasks are usually subject to each individual's personal experience—thus, developers rely on their own creativity and discretion. Moreover, names are oftentimes coined in a rush while solving an issue, hence the brain is normally under pressure and will most likely not put too much effort when choosing a name [36].

Several major companies try to impose a set of guidelines and standards on specific software languages recommending how they should be written in various aspects; these are called *coding conventions* [2]. The purpose of adapting and utilising coding conventions is to improve readability, and consistency within the use of a software language to ease software maintenance. While following coding conventions are recommended and have a large impact on software quality and maintenance [15], compilers do not require them.

Naming conventions are among the many coding conventions. They make programs easier to understand and facilitate comprehension by specifying identifiers for variables, functions, constants, types, and other entities. For example, naming convention for method names in Java established by Java community [69] will be expressed like this:

```
greet();
greetDog();
getDogName();
```

Here method names should include *verbs*, and every first letter of a word in the name should be capitalised, except from the very first letter of the name. Additionally, all spaces are removed[4] [46]. Similarly, one of Python's coding conventions is called *Python Enhancement Proposal 8*, also known as PEP 8, written by Rossum et al. [81]. This is one out of several PEPs that has been created. Below is an example of PEP 8's naming convention for method names.

[4]FORTRAN used to allow spaces in identifier names, saying "consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, [...] ." [40].

```
greet()
greet_dog()
get_dog_name()
```

This convention is similar to Java's convention. However, all letters are lowercase, and words are separated by a single underscore [81]. There are multiple choices of coding conventions available, making it hard for programmers to choose which convention is more superior than others, as well as be on the same convention when coding with peers.

The Hungarian notation, developed by Simonyi, is another naming convention that encodes the identifier's type and intention or kind into the name itself [75]. Simonyi described this as part of his doctoral dissertation [74], which later became the standard convention inside Microsoft [75]. The following illustrates an example of this convention.

```
bIsDog
nDogs
strDogName
```

The first name `bIsDog` is a boolean variable the second `nDogs` denotes the dog count, and the last `strDogName` is a string that represents the dog name. This convention was designed to not depend on any programming languages, thus can be used and found in various of languages. It is especially helpful for languages that do not explicitly declare or have much data types, like Basic Combined Programming Language (BCPL)[5] that is no longer commonly used. Including the types into the names can save programmers a lot of time when comprehending code, additionally can improve readability.

2.2.3 Name Quality

There are many different perspectives on what defines the quality of a name. Researchers in the field who studied software names do not all share the same view on this matter. Butler et al. [14] did an empirical study resulting in a collection with definitions of what makes bad names. Table 2.1 is a sample of this collection.

[5] "The most significant simplification is that BCPL has only one data type—the binary bit pattern—[...]" [71].

Name	Description	Bad name example
Capitalisation anomaly	*Identifiers should be appropriately capitalised.*	`pAgECounTEr`
Dictionary words	*Identifiers should be composed of words found in the dictionary and abbreviations, and acronyms that are more commonly used than the unabbreviated forms*	`strlen`
Excessive words	*Identifiers should not be composed of no more than four words or abbreviations.*	`convert_the_page _to_raw_int_bits`
Short identifier name	*Identifiers should not consist of fewer than eight characters, with the exception of:* `c`, `d`, `e`, `g`, `i`, `in`, `inOut`, `j`, `k`, `m`, `n`, `o`, `out`, `t`, `x`, `y`, `z`.	`name`
External underscores	*Identifiers should not have either leading or trailing underscore.*	`__count_`

Table 2.1: Sample of naming convention from Butler et al. [14].

The naming convention from Butler et al. go against the Hungarian notation from Simonyi. The name `bIsDog` will be seen as a good name according to the Hungarian notation, whereas according to Butler et al. this name is too short for an identifier name containing less than eight characters, hence will be considered as a poor name. On the other hand, Butler et al. and Allamanis et al. share some views when it comes to good naming practices. Allamanis et al. value consistency within code base [2], which Butler et al. seem to agree with based on the naming convention creation. Naming conventions are essentially about keeping code names *consistent* throughout the whole code base.

Feitelson et al., who did research on how to select better names, express that naming is problematic partly because names stem from natural languages that can be ambiguous [24]. According to the experiment in the study, when programmers chose a specific name for a variable—even though they all chose different names—the majority of their peer programmers would still understand the newly chosen name. Feitelson et al. called the occurrence, where most names have the same pattern with only a small change, for *name molds*. An example of a name mold is shown in Table 2.2. The word "treat" in Table 2.2 can additionally appear in both singular or plural form. There exist many types of name molds, however, involving less molds has higher chance for readability and code comprehension. Feitelson et al. developed a model involving a *three-step process* of how to name better [24]. Following the model can result in names with more use of concepts, that are longer and with higher quality. The three-step process model is as follows:

- Select the concepts to include in the name.

Mold
treat
max_treat
max_treat_per_month
treat_per_month
max_monthly_treat
max_month_treat
max_treat_num
treat_max_num
max_number_of_treat
max_num_of_treat
max_treat_amount
max_acc_treat
max_allowed_treat
monthly_treat_limit

Table 2.2: Name mold example [24].

- Choose the words to represent each concept.
- Construct a name using these words.

Gresta et al. carried out an empirical study of naming practices in Java projects, where eight identifier categories were found from 40 open-source projects [35]. The study indicate that most identifier names are based on the context, hence are context-specific. Binkley et al. suggest that a good name is a name that has a limited length, as well as limited vocabulary [11]. One of the reasons for this is due to longer names taking a toll on the programmer's memory. Meanwhile, Hofmeister et al. express that abbreviations and names consisting of a single letter, such as Hungarian notation, are poor naming choices and will hinder code comprehension [37].

2.3 Approaches From Linguistics

The study of language scientifically is called *linguistics*. It is the formal studies of the structure, use, and meaning of language [85]. Linguists analyse all aspects of language, including cognitive and social aspects, as well as the history of, connection between and changes within language families [64]. Linguistics is a multidisciplinary field which liaises closely with other disciplines and fields in natural sciences, social sciences and formal sciences.

There are bridges between linguistics and neuroscience—the study of the nervous system and human brain—especially studies of brain structure and brain function [77]. It is also a multidisciplinary field that draws with other disciplines such as linguistics, computer science, psychology, sociology and more. Hence, neuroscience is relevant for linguistics when studying how languages are used, impact and stored to the human brain. Having clear and consistent names that follow a systematically formatting rule of how to name identifiers has a higher chance of helping the brain to cognitively process the code [36]. Butler et al. explored the influence names have to the cognitive processes when comprehending programs [16]. Naming conventions show to require less cognitive processing, according to the study from Butler et al., thus aid in code readability and comprehension. Identifier names such as `prntmn` is more challenging to understand than names like `printMethodNames` where it is clearer with all the words spelled out, thus creates less mental effort despite being much longer. According to the study by Allamanis et al., accurate and descriptive names are vital for readability of code, especially from a cognitive perspective [3]. Poor naming choices and linguistic antipatterns, such as misleading names, have higher chance leading to defects in the software. Arnaoudova et al. describe linguistic antipatterns as *recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity*, where 17 types of linguistic antipatterns are related to inconsistency [5].

2.3.1 Natural Language Processing

Natural language processing (NLP) is a field in computer science that also connects with linguistics [60]. Both computer science and linguistics study languages, such as syntaxes, semantics and pragmatics. Not to mention, both fields are also used by humans. NLP brings the two together by studying how languages are processed by computers. Linguists use computers to comprehend and analyse languages, meanwhile programmers use linguistics to improve their programming [10, 19].

There are a lot of use cases for NLP in artificial intelligence, machine learning and deep learning [19]. Many programming languages provide a great variety of tools and libraries for solving NLP tasks, such as programming language Java and Python. NLP supports the development in intelligent virtual assistants like Apple's Siri, Amazon's Alexa, Samsung's Bixby and Google Assistant [6, 44, 45]. Virtual agents use speech recognition to recognise and process human speech commands into written text when humans talk to them, and then generate an appropriate natural language response back. Another software programme NLP works behind is the *sentiment analysis* of social media.

NLP can reveal the emotions that lay in the language used in posts, reviews, reactions and messages by consumers of social media. These emotions are insights of veiled data that can be used for business purposes such as advertisement campaign, product feedback, audience targeting and more. There are also NLP solutions in search engines to, for instance, collect the *synonyms* of the words users search to provide the best solution back as possible.

To solve complex NLP tasks, the tasks are usually broken down to more specific techniques, where each technique works on its own way of processing the software language. Examples of such techniques will be discussed in the following.

Part of Speech Tagging

Part of speech tagging refers to classifying each word of a sentence into categories with similar grammatical properties, based on the context and role of the word explaining its usage [58]. These part of speech categories have many names, including the term *word class*, where the English language has 36 word classes according to the *Penn Treebank Project* [54]. Some examples of these are: conjunction, cardinal number, determiner, foreign word, preposition, adjective, noun, pronoun, adverb, interjection, and verb. part of speech analysis must be done separately for each individual language, as most languages contain different amounts and types of classes. Most languages have classes noun and verbs, however, some languages have several variations of the same class[6]

A single word can have several different meanings depending on the sentence it is in, ergo can serve as multiple classes. To demonstrate this the word "*watch*" can be a noun meaning a timepiece usually worn on the wrist in a sentence like "*she was gifted a watch*", or it could be a verb implying someone to look out, be alert, look at or observe something like for instance "*can you watch my dog? He is dangerous, so watch out*". The word "*well*" can be classified up to at least five classes; verb, adverb, noun, adjective and interjection. The classes are shown as tags when analysing words, which means the sentence "*print the bad method names*" can result in the part of speech tags "VB DT JJ NN NNS", where VB stands for verb, DT for determiner, JJ for adjective, NN for singular or mass noun and NNS for plural noun.

[6]An example of this is the Japanese language with at least two categories of the adjective class [63].

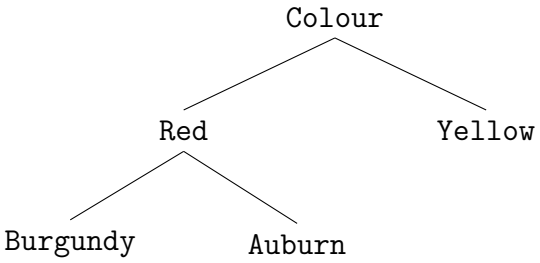

Figure 2.2: Hyponyms and hypernyms example.

Synonyms, Hyponyms, and Hypernyms

Most words have synonyms—semantically equivalent word or expression—which are syntactically different from the original word. In most cases, replacing a word in a sentence with its synonym will not significantly—if at all—affect the meaning and essence of the original sentence. However, not all words that are synonyms will make sense in all types of sentences, hence it is not a bulletproof check. Some synonyms of "*create*" are "*generate*", "*build*" and "*design*", and the sentences "*create new language*", "*generate new language*", "*build new language*", and "*design new language*" all yield either the same meaning (for a non-computer scientist), or four different meanings (for example, for a software language engineer).

Furthermore, synonyms can be *hyponyms* or *hypernyms*. Hypernyms are words that are general categories for specific words. Those specific words in a hypernym category are called hyponyms. Figure 2.2 shows an example of hyponyms and hypernyms. From the example, the word "*Colour*" is a hypernym, and two of its hyponyms are "*Red*" and "*Yellow*". These two hyponyms are considered co-hyponyms, since they belong in the same hierarchy level and share the same hypernym. Hyponyms can be hypernyms too for the hyponyms categorised below them in the hierarchy. This means that if the hyponym word "*Red*" has "*Burgundy*" and "*Auburn*" as hyponyms, then "*Red*" is also a hypernym for the two co-hyponyms below it.

Sentiment Analysis

Sentiment analysis can unveil subjective information, such as the attitude and emotions, of sentences [25]. It is often used to analyse digital reviews or feedback and categorise them, as well as for easier recognition of what public opinions says from social media. There are usually three common accepted sentiments: *positive*, *neutral* and *negative*.

Some words and sentences can have opposite or several sentiments depending on how it is expressed and worded. A word such as "*print*" is neutral, but can result in a different sentiment that is either negative or positive when combined with other words. For example, "*print error*" typically gives off a negative sentiment, whereas "*pretty print*" will result in a positive sentiment. Just as how hard it is for humans to detect the right emotions behind natural speeches and texts, it can also be as challenging to programmatically draw out the right sentiments from natural languages.

Linguistics involve many more subfields and areas. These can be categorised into two fields: theoretical linguistics and applied linguistics [52, 72]. Theoretical linguistics is mainly about constructing linguistic theories. Meanwhile, applied linguistics is more practical compared to theoretical linguistics. This field utilises the knowledge of languages, such as how languages are learned and used, for solving language related problems in the real world. Language related problems are issues that can arise when, for example, studying how computers process and analyse natural languages. This involves NLP that is mentioned above with a few NLP techniques explained.

2.4 Domain-Specific Languages

Domain-specific languages (DSLs) are programming languages, or specification languages, that specialise in specific domains with fixed set of conditions [31]. The domain of a DSL can be expressed as the set of problems it can model and solve. It is not possible to use it like a programming language such as Java, Haskell or C, due to DSLs' focus on a specific domain. Examples of some known DSLs are HTML (for web pages), SQL (for querying relational databases), CSS (for style sheets) and LaTeX (for writing documents).

Languages like JavaScript, C++ and Python are considered as GPLs. There are advantages and disadvantages with both DSLs and GPLs. Compared to a GPL, a DSL favors its intended domain more by being exceptionally specific and sacrificing generality and flexibility. Additional benefits of using DSL is making code easier to read, hence, mistakes and errors are more easily preventable [84]. Domain-specific concepts can enable abstractions and model assumptions, improving analysing and designing DSL applications. It is important being able to define the scope of the DSL, to not create an unnecessary big and complex language. This requires knowing what to add and what to keep out of the DSL. The DSLs' limited scope makes it easier for users to learn the language, in contrast to a GPL. However, committing to building or using a DSL can

involve adversity, especially when one is not comfortable using DSL or know its concepts and principles. In many cases, DSLs are used by domain experts within a non-software field, where the users do not need to have any other technical knowledge outside of the domain considering it is domain focused.

The execution engine of a DSL can either be an interpreter or a compiler, also known as code generator [84]. DSL with an interpreter reads in the DSL script and executes it at run time. In contrast, a compilation generates the DSL programme often first into a high level language source code like Java, and then runs it directly on the targeted platform.

DSLs are mainly divided into two groups: internal and external DSLs. Some DSLs are embedded into GPLs, these are classified as the internal DSLs that work within another programming language. Naturally, internal DSLs are limited to the compiler or interpreter, syntax, model and concepts of the host language [80]. This can be seen as an advantage as it can reduce the cost of building. Internal DSLs act quite similar to application programming interfaces (APIs), and at times it can be difficult to distinguish the two, thus it is also referred as *fluent interfaces* [84, 31]. GPLs that tend to use internal DSLs are for example Lisp, Ruby, Haskell, Groovy and Python. On the other hand, external DSLs are mainly built from scratch. This way, the external DSLs parses independently and is not tied to any host GPL. Everything can be customised from the parsing to the execution of a programme. As beneficial as it sounds, it also increases the cost of building the DSL. A lot of time is needed to create a well-designed language that is so fully customised from scratch. However, fortunately external DSL have tools that diminish the time invested into building it. Such tools can be helpful integrated development environments (IDEs) that are supportive and aware of the language's needs. Many programmers today use Eclipse as their preferred IDE [9]. Using IDEs can enhance user experience and increases the chances of the DSLs to be embraced and successful. IDEs can support features like syntax highlighters, auto-code-completion, immediate feedback, hyperlinks, debugger, visualisations and more. These features have the potentials of making it easier for new programmers to learn, use, develop and maintain DSLs.

One of the important elements of DSLs—also found in other software languages—is the *concrete syntax*. Textual DSL, graphical DSL, symbolic DSL, tabular DSL or a mix of these are the main classes for DSL's concrete syntax, denoting the notation that users can express programs [84]. The most common type is the textual DSLs [31], which uses textual notations or syntax. A Graphical DSL requires help from a tool such as language workbenches. Language workbenches are tools that work well as meta-languages in their normally own powerful IDEs to ease the cost of creating DSLs, as well as can efficiently be

integrated [23]. A few examples of language workbenches are: Eclipse Xtext[7], JetBrains MPS[8], Spoofax (SDF/Stratego)[9] and Eclipse Sirius[10]. Eclipse Xtext and Spoofax are textual DSLs [22, 43], meanwhile JetBrains MPS is projectional DSL [83] and Eclipse Sirius is graphical DSML[11] [82]. JetBrains MPS supports a mixture of multiple types of notations (textual, symbolic, graphical and tabular), hence, it is called a projectional DSL. All these mentioned language workbenches are compatible with Java as the target programming language for model transformation and code generator.

2.5 Eclipse Xtext

Eclipse is an integrated development environment (IDE), widely chosen as the Java IDE of choice [33]. Meanwhile, Xtext is an open-source Eclipse Framework that can be utilised for implementing DSLs, editor plugins and text editors for web browsers with the Eclipse IDE integration and IntelliJ IDEA [9]. Xtext makes it possible to design languages with a full infrastructure as a complete programming language and an IDE with features. Every single aspect of the implementation can be customised by the programmer, although Xtext also includes default implementations for aspects not needed to be customised.

The grammar language of Xtext is what defines Xtext as a textual language [8]. A grammar is a set of rules specifying the correct structures of language elements, like for example how they should be formed and expressed in a language [9]. It can also be considered as a description of the concrete syntax of a software language. Xtext automatically handles most of the building and creation of the abstract syntaxt tree (AST), hence only the grammar specifications is needed to start implementing the DSL. The grammar can be simple, and as long as there is a grammar, Xtext will generate and arrange the rest of the concepts. It also supports reuse of grammars that have already been specified before [8]. This means that when another grammar is included into a language, it will be possible to refer to the rules of that grammar, as well as overwrite its rules. In other words, the terminal rules that are declared holds higher priority than the terminal rules that are imported. When the grammar is specified, the code for the lexer and parser get automatically generated, and the DSL is ready for use.

[7] `https://www.eclipse.org/Xtext/`
[8] `https://www.jetbrains.com/mps/`
[9] `https://www.metaborg.org/en/latest/`
[10] `https://www.eclipse.org/sirius/`
[11] `https://ieeexplore.ieee.org/servlet/opac?bknumber=7043955`

Java is compatible with Xtext and can be used for customising the implementation of DSLs, although, Xtext promotes the use of Xtend [9]. Xtend is a programming language that resembles Java, a dialect of Java, that can optimise code generation. It is allegedly easier to use than the standard Java, due to its greater flexibility and improvement on many aspects, yet in the end translates to Java source code. Xtend is fully inter-operable with Java, considering it is similar and supports every aspect of Java, especially the type system. This statically typed language also ensures much more clearer and cleaner programs.

When creating an Xtext project, the Xtext wizard will generate a few additional projects including the one initially created. For example, the following is the specifications of the project creation:

- Project name: `org.example.entities`
- Name: `org.example.entities.Entities`
- Extension: `entities`

Then, the following Xtext projects will be created:

- `org.example.entities`: The main project (including the grammar specifications and components that are independent from the UI).
- `org.example.entities.ide`: The IDE (Include components related to the UI and independent from Eclipse).
- `org.example.entities.tests`: The tests (including JUnit tests that do not depend on the UI).
- `org.example.entities.ui.tests`: The UI Tests (including JUnit tests that depend on the Eclipse UI).
- `org.example.entities.ui`: The UI (including components related to the Eclipse UI).

Xtext generates an editor that is by default an Eclipse plugin. This editor can be customised, in addition to having the possibility of creating a rich client platform (RCP). In short, RCP application uses the Eclipse framework to create a simpler version of Eclipse itself, but independently supporting the implemented language and does not include unnecessary features originally from Eclipse [55]. The Xtext editor can generate editors for IntelliJ IDEA and also includes a web editor support. Integrating text editors in

web application was offered since version 2.9 [32]. According to Eclipse Xtext documentations [28], JavaScript has been used to implement the text editors, and services like code completion is handled through HTTP requests to a component on the server-side. Xtext's supported clients are three JavaScript text editor libraries: Orion[12], Ace[13] and CodeMirror[14].

2.6 JavaParser

JavaParser[15] is an open-source library that allows interactions with Java source code through a Java abstract syntax tree (AST) [62]. The library helps parsing source code and provides aid to navigate around the AST, giving programmers the ability to traverse the code without having to write the tree traversal code from scratch. JavaParser can also unparse—that is, pretty print—an AST back to Java source code. The fundamental feature of the library is to provide programmers the capability of building their own code by manipulating the structure of the source code. The library is a strong tool to analyse, transform and generate code base up to Java 12.

To understand how JavaParser works, one must understand the notion of an abstract syntax tree first. Simply explained, AST in Java can be seen as abstract objects representing the source code in an environment in Java. Figure 2.3 illustrates a general AST example. The object representations are represented as a tree, starting with a single point that is considered as the *root* of the tree. From this root and downwards, there are *branches* forming out independently representing a code statement. At the end-tip of every branch of the AST there are *leaves*. A leaf is the last object with no following objects forming out of it, representing the last code statement in this route. Similarly to a real tree, there are many branches forming out independently from the main source, the roots. From there, more branches can grow out from a branch, but once there is a leaf, that is the last terminal element of a tree.

Variable references and method calls tend to come from various parts of the source code. Figure 2.3 shows an example where there is a method call `aMethodACall` coming

[12] `https://projects.eclipse.org/projects/ecd.orion`
[13] `http://ace.c9.io/`
[14] `http://codemirror.net/`
[15] `https://javaparser.org/`

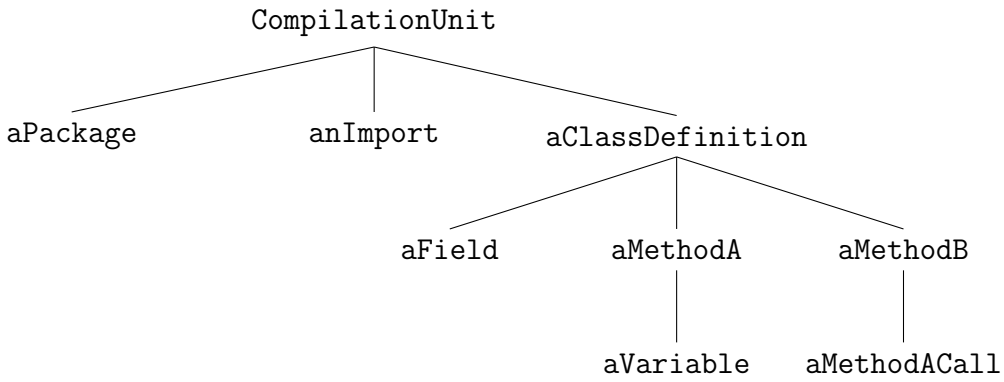

Figure 2.3: JavaParser library AST example.

from branch `aMethodB`; however, is related to another branch `aMethodA`. These connections are not picked up by the syntax trees built with JavaParser. To detect related elements and create relationships across branches, a symbol solver is needed. The JavaSymbolSolver[16] is integrated in JavaParser for adding connections between objects relating to each other.

There are many ways of creating an AST from the source code. One way of generating an AST, is to parse the source code and output it as a YAML[17] file in the command line. Figure 2.4 demonstrates an example of how this can be implemented. The `compilationUnit` is the root of an AST, and to produce an AST, the `StaticJavaParser` will parse through a file with the help of a `FileReader`. Then, the `YamlPrinter` class from JavaParser is utilised to print out the AST of the source code.

```
1 public class printAST {
2     public final static String FILE_PATH = "GoodNumber.java";
3     public static void main(String[] args) throws IOException {
4         CompilationUnit compilationUnit = StaticJavaParser.parse(new
              ↪ FileReader(FILE_PATH));
5
6         YamlPrinter printer = new YamlPrinter(true);
7         System.out.println(printer.output(compilationUnit));
8     }
9 }
```

Figure 2.4: JavaParser YAML printer implementation in the `printAST` Java class.

The AST is being generated based on the Java file called `GoodNumber`. Figure 2.5 shows the content of this file. It is kept short with one method for the simplicity of the

[16] https://github.com/javaparser/javasymbolsolver
[17] https://yaml.org/spec/history/2001-08-01.html

```
1 public class GoodNumber {
2     public int getNumber() {
3         return 8;
4     }
5 }
```

Figure 2.5: An example Java source code file called `GoodNumber`.

AST that will be generated, as ASTs easily grow bigger the more code that is involved. As a result, Figure 2.6 shows the output from parsing and printing the `GoodNumber` source code to a YAML file.

The YAML output is normally a sufficient representation of the AST, as it shows everything similarly according to the hierarchy system. However, it can also be further developed to an actual tree representation as Figure 2.7 shows, resembling the previous AST mention from Figure 2.3.

Analysing Java source code becomes low-effort with this library. Figure 2.8 demonstrates an example of an analysis of the `Example` Java file. This demonstration shows how method names can be printed with the help of the library. Similarly to Figure 2.5, the implementation starts with the `compilationUnit` that parses through the Java source code. Additionally, the class `MethodNamePrinter` extends the abstract class `VoidVisitorAdapter<Void>`. Here the implementation of the `visit` is overriden with the `MethodDeclaration` class, which is the preferred class when analysing methods. Eventually, `super` is called to make sure the respective child nodes are visited and to prevent any unwanted performances. Then, the method names are retrieved with the `getName` method from `MethodDeclaration`. And at last, the result will print all the method names in the `Example` Java file. For more details, the JavaDoc[18] has further information of the packages and use cases from the library.

[18] `https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/latest/index.html`

```
---
root(Type=CompilationUnit):
    types:
        - type(Type=ClassOrInterfaceDeclaration):
            isInterface: "false"
            name(Type=SimpleName):
                identifier: "GoodNumber"
            members:
                - member(Type=MethodDeclaration):
                    body(Type=BlockStmt):
                        statements:
                            - statement(Type=ReturnStmt):
                                expression(Type=IntegerLiteralExpr):
                                    value: "8"
                    type(Type=PrimitiveType):
                        type: "INT"
                    name(Type=SimpleName):
                        identifier: "getNumber"
                    modifiers:
                        - modifier(Type=Modifier):
                            keyword: "PUBLIC"
            modifiers:
                - modifier(Type=Modifier):
                    keyword: "PUBLIC"
...
```

Figure 2.6: JavaParser YAML output of the `GoodNumber` Java class.

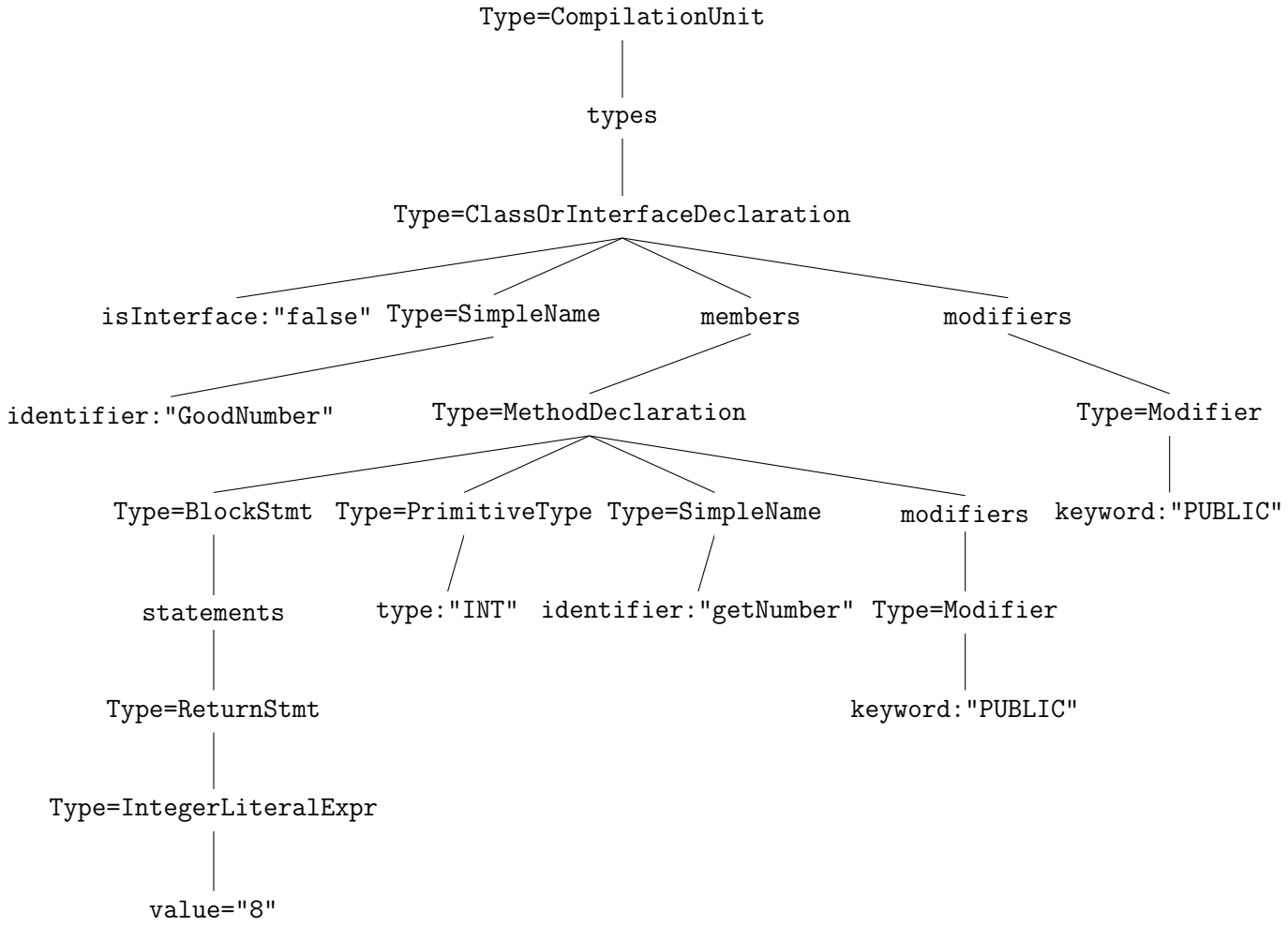

Figure 2.7: JavaParser AST representation of the `GoodNumber` Java class.

```java
public class MethodVisitor {
    public final static String FILE_PATH = "Example.java";

    public static void main(String[] args) throws IOException {

        CompilationUnit compilationUnit = StaticJavaParser.parse(new
            ↪ FileReader(FILE_PATH));

        VoidVisitor<Void> methodNameVisitor = new MethodNamePrinter();
        methodNameVisitor.visit(compilationUnit, null);
    }
}

class MethodNamePrinter extends VoidVisitorAdapter<Void>{
    @Override
    public void visit(MethodDeclaration method, Void arg) {
        super.visit(method, arg);
        System.out.println("Method Name: " + method.getName());
    }
}
```

Figure 2.8: JavaParser library code example of printing all method names.

Chapter 3

Tool for Analysing Names

In this chapter, the developed tool for analysing names will be presented, as well as the usage of the tool. To start with, the idea of the name analysis tool will be described. Then, several detailed examples on how to use the tool from the user's perspective will follow.

3.1 The Analysis Tool

The idea of the analysis tool is to analyse names of various identifiers, including variables, methods and classes, by checking whether the identifiers' implementation satisfy certain requirements according to their names. For instance, there is a requirement for all method names that has the word `find` to include local variables and contain loops. Now, the tool will be given a file with Java source code, and check if there are method names with the word `find`. When the right method names have been found, the method's body will be checked for local variables and loops. In the end, a report of the findings, whether the identifiers satisfy the requirement or not, will be presented to the user after the analysis is completed.

3.1.1 Requirements

The user can specify the requirements using a DSL that has been designed and implemented. Figure 3.1 illustrates how these requirements can be specified and written using the DSL implementation. There are three sections of the specification that are important: the declarations, the rules and the cases.

```
declarations
    // custom declarations
    Parameter <-> "com.github.javaparser.ast.body.Parameter"
    WhileStmt <-> "com.github.javaparser.ast.stmt.WhileStmt"
    // pre-defined declarations
    method <-> "com.github.javaparser.ast.body.MethodDeclaration"
    variable <-> "com.github.javaparser.ast.body.VariableDeclarator"

rules
    // pre-defined rule
    def declare callsMethodWithSameName
    // custom rule
    def hasNoParameter for method {
        filter Parameter
        such that size < 1
    }

cases
    // requirement for method with an example condition
    case for method "hello"[neutral].NN {
        always
            hasNoParameter
        often
            callsMethodWithSameName
    }
```

Figure 3.1: Requirement specifications example.

Declarations

To start with, the user needs to define declarations for identifier types. This is due to the fact that this tool uses JavaParser to traverse the Java source code. Therefore, all classes that are relevant and will be used need to be declared with a JavaParser class path. The declarations can be of two kinds: pre-defined and custom. Below are the examples of the pre-defined declarations.

```
declarations
    package <-> "com.github.javaparser.ast.PackageDeclaration"
    class <->
        ↪ "com.github.javaparser.ast.body.ClassOrInterfaceDeclaration"
    interface <->
        ↪ "com.github.javaparser.ast.body.ClassOrInterfaceDeclaration"
    method  <-> "com.github.javaparser.ast.body.MethodDeclaration"
    variable <-> "com.github.javaparser.ast.body.VariableDeclarator"
```

This is the declarations specifically for the types of the identifiers, such as packages, classes, interfaces, methods and variables. The syntax for the other declaration type is as follows[1].

```
declarations
    <Class Name> <-> "<Class Path>"
```

This declaration type is custom, and can be used for any other types. The `<Class Name>` must be a unique name, and the `<Class Path>` is the path to a class that is from the JavaParser library. An example of the class path to the parameter class is `"com.github.javaparser.ast.body.Parameter"`, which represents the parameters of a method or lambda. It is possible to declare any class type as long as it is visitable and there exist a class path for it in the JavaParser library[2].

Rules

To check the identifiers' bodies, the `rules` of requirements need to be defined. As an example, when there is a requirement that the method name has no parameters, a formal

[1] A formal grammar for the DSL is given in Appendix C

[2] An exhaustive list of possible types is available at `https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/latest/index.html`.

definition of having no parameters needs to be given. The rules specify what classes to filter out and look for in the body of an identifier, and then conclude if the classes are present in the body or not. This tool has one rule that is already defined and can not be modified, and that is the rule name `callsMethodWithSameName`. The `callsMethodWithSameName` checks if any method calls in a method has the same name as the method itself. To enable the usage of this rule, the following declaration is used:

```
rules
    def declare callsMethodWithSameName
```

The syntax of rule declarations is as follows.

```
rules
    def <Rule Name> for <Identifier Name> {
        filter <Class Name> (| <Class Name>)*
        such that size <Comparison Operator> <Number>
    }
```

Every rule has a unique `<Rule Name>`, identifying which rule it is. The rule should state what identifier it applies for at the `<Identifier Name>`. This `<Identifier Name>` should be one of the already declared identifier types from the `declarations`. Following this, the rule will consist of what `<Class Name>` it should filter, which should also be among the same class types declared from `declarations`; here it is possible to filter either one or several classes. As an example, to filter three classes, the following expression is used: `filter WhileStmt || ForEachStmt || ForStmt || DoStmt`. Finally, the size of the filtered list will be compared according to what the specified `<Comparison Operator>` and `<Number>` are. The following Table 3.1 shows the valid comparison operators, and `<Number>` can be any chosen number.

Comparison Operator	**Meaning**
`>`	*Greater than.*
`<`	*Less than.*
`==`	*Equal to.*
`<=`	*Less than or equal to.*
`>=`	*Greater than or equal to.*
`!=`	*Not equal to.*

Table 3.1: Valid comparison operators.

Below are three examples of rules for the identifier method showcased:

- **Number of parameters**: Method having a certain number of parameters.

```
def hasThreeParameters for method {
    filter Parameter
    such that size == 3
}
```

- **Return type**: Method returning a certain type, e.g. returning a void.

```
def returnsVoid for method {
    filter VoidType
    such that size > 0
}
```

- **Containing statement**: Method's body containing a certain type of statement, e.g. loop statements.

```
def containsLoop for method {
    filter WhileStmt || ForEachStmt || ForStmt || DoStmt
    such that size > 0
}
```

Cases

When the declarations and rules have been defined, various cases can then be specified with conditions and requirements. The `cases` filter identifier names with a certain condition, and then checks if the identifier's body satisfy a chosen frequency of a rule. In other words, this is where users can specify requirements when analysing the source code. The syntax of cases specifications is as follows.

```
cases
    case for <Identifier Name> <Condition> {
        <Frequency>
            <Rule Name>
    }
```

Similarly to `rules`, the cases use the `<Identifier Name>` from `declarations` to specify what identifier each case is relevant for. Then, there is a `<Condition>` that filters the names of the relevant identifiers. The condition is a sequence of queries for constituents

of names. To begin with, an example is that the tool should analyse methods with the name `"print"`. The condition will look like this:

```
case for method "print" {
    ...
}
```

This name is a string, hence it can be identified through matching strings. When processing texts, it is very common to match texts with each other. *Regular expressions* [78] provide a way for analysing, matching and manipulating strings. This usually contains a set of characters that forms a pattern, which will then be used to effectively find any matching strings to the pattern. Being able to match and detect strings is useful for extracting out the relevant strings that are important for further use.

However, not all programmers choose their method names similarly using the exact same words even though the method bodies have the same concept and structure. Additionally, there exist several common words used for the same concept, like for instance when obtaining the length or size, the word "size" and "length" are often times used interchangeably. To solve the issue, where the user does not want to settle for a specific word, synonyms can be used. Including the synonyms of a specific string can be expressed by preprending the string with a hash sign. Continuing the example from above, the following shows the condition including the hash sign representing the synonym of the string:

```
case for method #"print" {
    ...
}
```

This condition allows synonyms of "print", which includes the words "engrave", "inscribe" and "dump".

A word may also be a certain part of speech. As explained in Section 2.3.1, POS tagging is the role of a word in a sentence. This is useful when there is no specific words to specify, but the user still wants to have a type of word in a specific place in the name. For example, wanting to include a cardinal number, an adjective and a noun after the word "print", but not needing to specify what specific cardinal number, adjective or noun to involve. The *Penn Treebank Project* presents 36 part of speech tags and describes what each tag stands for [54]. For instance, the part of speech tag for a cardinal number is CD, for a singular noun is NN and for an adjective it is JJ according to Figure 3.2. Continuing

Tag	**Description**
CC	Coordinating Conjunction
CD	Cardinal number
DT	Determiner
EX	Existential *there*
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	particle "*to*"
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

Table 3.2: Part of speech tags used in the Penn Treebank Project [54].

the example from above, the following shows the condition including the three mentioned part of speech tags:

```
case for method #"print".CD.JJ.NN {
    ...
}
```

In between every word in the condition, there is a period (".") in the middle separating each word in a method name.

It is not always clear solely from the name of an identifier what sentiments or emotions are behind the chosen words. Especially with the possibility of part of speech tagging, it can be convenient knowing, for example, if the JJ (adjective) should have a negative or a positive sentiment. The desired sentiment can be specified with square brackets around the word or part of speech tagging that wants to have a sentiment involved. Continuing the example from above, the following shows the condition including a sentiment analysis:

```
case for method #"print".CD.JJ[negative].NN {
    ...
}
```

To make the condition more flexible and feasible, it is possible to add cardinality modifiers to the strings, synonyms and part of speech tags. The accepted cardinality modifiers are the following:

- `*`: has the cardinality of zero or more elements
- `+`: has the cardinality of one or more elements
- `?`: has the cardinality of zero or one element

Additionally, a part of the name can consist of several choices and not be bound to a single type of string, synonym or part of speech. This can be expressed by putting the alternatives together in parentheses with a pipe symbol separating them. An example to illustrate this is: `"(NN|VB|JJ|NNS|UH)"`. Continuing the example from above, the following shows the condition including cardinality modifiers and the pipe symbol.

```
case for method #"print".(DT|CD)?.JJ*[negative].(NN|VB|JJ|NNS|UH)* {
    ...
}
```

The condition can be represented as a regular expression. This one particular example above will match all names that starts with the word "print" or a synonym of "print", followed by an optional determiner or a cardinal number, and then include an optional negative single or multiple adjectives, ending with none or several noun, verb, adjective or interjection. An example of a name that will match this pattern of regular expression is `"printTheBadMethodNames"`, which will satisfy the `"print"`, `DT`, `JJ[negative]`, `NN` and `NNS`. Table 3.3 demonstrates samples of conditions and corresponding samples of names that satisfy the conditions.

Condition	Name
`"print"`	print
`#"print"`	print; dump; engrave
`"print".DT.JJ.NN`	printTheGoodDocument
`"print".DT.JJ[negative].NN`	printTheBadDocument
`"print".(DT|CD)?.JJ*[positive].(NN|VB|NNS|UH)+`	printFiveGoodMethodNames
`(NN|NNS)*.#"create".(DT|CD)?.JJ*[neutral].NN`	generateAnAlgorithm
`(NN|NNS)*.VB.CD?.JJ*[neutral].(NN|NNS|UH)*`	botPrintsTwoShortLists
`(NN|NNS)*.VB.(DT|CD)?.(NN|VB|UH)*`	botsGreetHelloWorld
`NN[positive]?.VB.(DT|CD)?.JJ*.(NN|VB|UH)*`	size
`NN*.VB+.JJ+.(NN|VB|UH)+`	findPreciseVariableName
`"create"."new"?.(NN[positive]|NNS[negative])`	createNewDream
`VB+.(DT|PDT)?.(NN|NNS)+.TO.(NN|NNS)+`	checkAllNamesToRequirements
`VB.(NN|NNS).IN?`	getSentimentOf
`VB.(NN|NNS)."of"?`	getSynonymsOf
`(NNS|NN[neutral])+."checker"`	ruleChecker

Table 3.3: Name condition and corresponding name examples.

After specifying the condition, the `<Rule Name>` are the requirements to be checked for the names that match the condition, whereas the `<Frequency>` implies on how often the `<Rule Name>` is expected to occur. The various frequency values are an inspiration from the study *The Programmer's Lexicon, Volume I: The Verbs* by Høst and Østvold [39]. This is terminology from the lexicon that describes the quantile of the attributes. An overview of all the phrases is shown in Table 3.4.

3.1.2 Interactive Command Line

There is an interactive command line that handles the interaction between the user and the tool. Moreover, this is where the user can create the connection between the specified requirements and the raw Java source file from GitHub. When running the analysis tool,

Phrase	**Meaning**
Always	*The attribute value is always 1.*
Very often	*The name is in the high extreme quantile.*
Often	*The name is in the high quantile.*
Rarely	*The name is in the low quantile.*
Very Seldom	*The name is in the low extreme quantile.*
Never	*The attribute value is always 0.*

Table 3.4: Lexicon Terminology [39].

a list with instructions will appear in the command line as shown in Figure 3.2. Three choices of actions are presented in the instruction list; the action `P`, the action `I` and the action `Q`.

To choose the first action, the letter `P` needs to be entered as an input. Figure 3.3 shows what happens in the command line when choosing `P`. It requires a file path to an XML file that consists of the requirements defined by the user. This XML file is generated by the implemented DSL when the users specify the requirements. Enter the file path, and the first action is completed, as well as the `P` action will be removed as it is fulfilled.

Now, assuming the first action is achieved, start on the second action by entering the letter `I` as input. Figure 3.4 shows what happens in the command line when choosing `I`. It requires a raw GitHub link to the Java source code file. This is the file that will be analysed, and that will be given a report and feedback on after the analysis is finished. Enter the required path file, and the tool will start analysing immediately, considering both top actions are fulfilled.

Lastly, the last choice of actions is `Q`, and choosing this will terminate the currently running interaction in the command line. There is no strict order of which action should be completed first. The user can choose to start with importing the raw Java source file link, and then finish off with passing the XML file with requirements. As long as both top actions are done, the progress of analysing will start, and shortly after a report of the analysis will be presented.

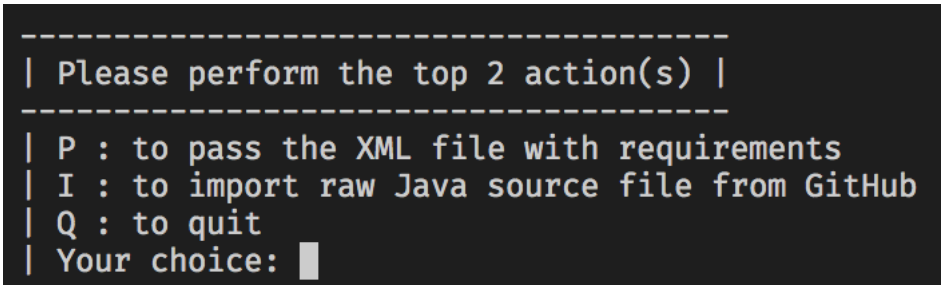

Figure 3.2: Interactive command line.

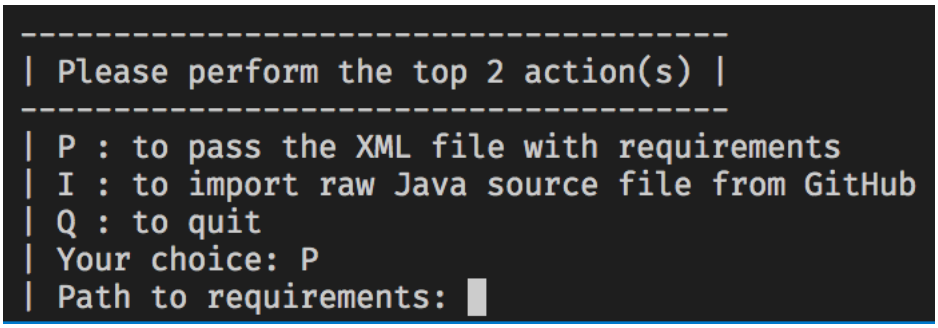

Figure 3.3: Interactive command line—choice: P.

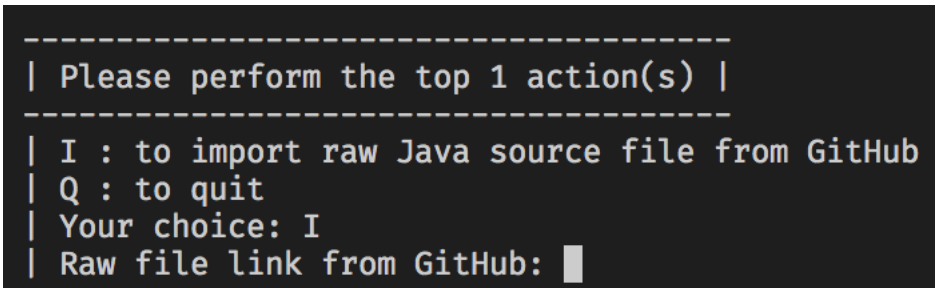

Figure 3.4: Interactive command line—choice: I.

Chapter 4

Implementation

In this chapter, the implementation details of the analysing tool will be presented. The overall software architecture will be described. Then, a technical descriptions of the grammar and code will be explained and code samples shown.

4.1 Architecture

The architecture of the work in this thesis is illustrated in Figure 4.1. There are several components involved, where six of them are grey and one has double borders. The grey with dashed border components represent tools and languages that can be changed out and replaced or extended. Alternating them out with corresponding tools shall not change the underlying main structure of the work. The double border component on the other hand is the most important component where the components meet, are connected and where the analysis is executed.

It first starts with the DSL that has been developed, where anything can be specified as requirements by the user using the implemented DSL. In this case, the lexicon from the Programmer's lexicon [39] is used as an inspiration for requirements to implement. The requirements are implemented in the Eclipse Xtext-based IDE. Moreover, when the requirements have been implemented in the IDE, relevant conditions will be checked for synonyms with the help of the synonym analysis library, leading to the generation of an extensible markup language (XML) file. This XML file is required to be generated for the requirements to be transferred over to Java. In the main component called `Name Analyser`, there are five components connected to it:

1. JavaParser library to parse through Java source code.
2. Java source code given by the user.
3. Sentiment analysis library that aids in detecting sentiments of words.
4. Part of speech (POS) analysis library that categorises each word to a class.
5. XML file with all the requirements.

The `Name Analyser` analyses names by utilising all five components. The JavaParser assist in parsing the Java source code, where the identifiers will be checked with the requirements in XML. Then, identifier names are processed with the help from sentiment and part of speech analysis libraries.

4.2 DSL and Java

There are two repositories hosting the work described in thesis. One is hosting the Xtext project with the implementation of the DSL[1], and the other is hosting the Java project that executes the analysis tool[2].

The linguistic approaches were performed by using three Java libraries: Extended Java WordNet Library (extJWNL), Stanford CoreNLP and Apache OpenNLP. To start with, the extJWNL is an open-source library that create, read and update dictionaries in the format of WordNet [7]. WordNet is an online lexical database registered trademark of Princeton University [57]. One of the linguistic approaches that extJWNL library supports is the analysis of synonyms, which is what this thesis uses the library for. Stanford CoreNLP is an NLP library [53], that supports this thesis with the sentiment analysis system [76]. At last, the Apache OpenNLP is another NLP library that assist the part of speech tagging in this thesis [30].

[1] `https://git.app.uib.no/Emily.Nguyen/nameanalyser`
[2] `https://git.app.uib.no/Emily.Nguyen/methodnameanalyser`

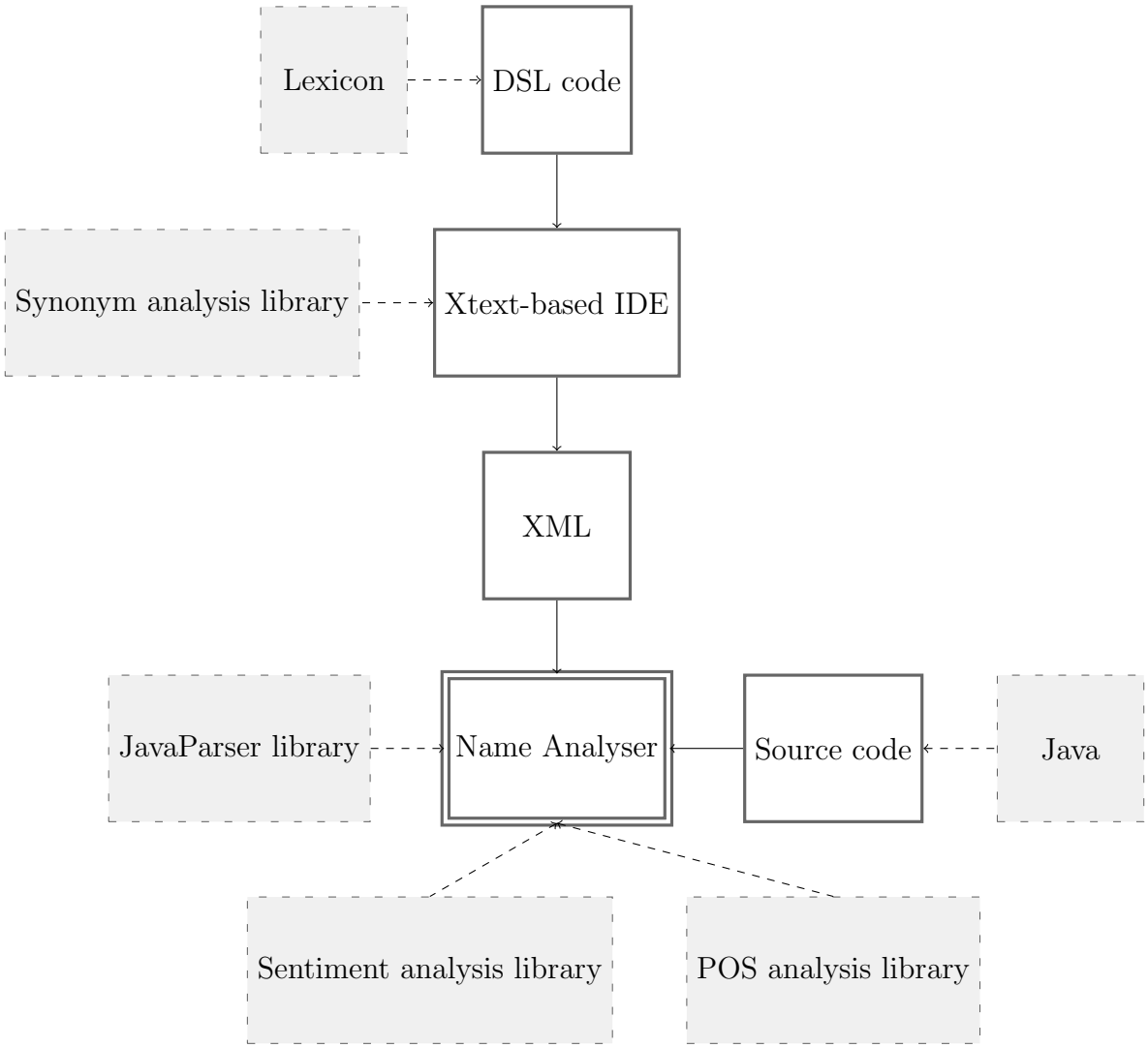

Figure 4.1: Overview of architecture.

4.2.1 Grammar

As explained in Section 2.5, a grammar is a specification of the concrete syntax of a software language. The grammar language of the built DSL is implemented in Xtext, using ANother Tool for Language Recognitiont (ANTLR)[3] to specify the grammar. Figure 4.2 shows a sample of this implemented grammar. Here, the syntax for the declarations, rules and cases explained in Section 3.1.1 are showcased. The full implementation can be found in Appendix C.

4.2.2 Code Generation

After the grammar is completely implemented, the code is then being generated with Xtend. Figure 4.3 shows a sample of how the code generator has been implemented. This implementation generates an XML file called `generatedRequirements.xml` with all the requirements transformed from the DSL to XML. As an XML file, the requirements are easier accessible and flexible to be used with Java. In the Java project, the XML file will be parsed and used to analyse the given Java source code. The full implementation of the code generator can be found in Appendix D.

4.2.3 Name Analyser

One of the most fundamental methods in the Java project of this work is the `ruleChecker`. Figure 4.4 shows the full `ruleChecker` method in Java. This method has four parameters: `caseTypeInput`, `listOfInstances`, `comparisonOperator` and `comparisonValue`. First of all, the `caseTypeInput` is a specific identifier implementation as a Node, for example the body of a method. Furthermore, the `listOfInstances` is a list of all the relevant instances of JavaParser class paths that have been included to this specific requirement. Lastly, the `comparisonOperator` and `comparisonValue` are as the names indicate, the comparison operator and the comparison value respectively specified from the requirement. This method loops through all the relevant JavaParser class paths, and attempts to check if there are any instances from the identifier implementation that will match the instances from the requirement. If there is a match, it will get accumulated into a collection that becomes a list of Nodes in the end of the iteration called `foundInstances`.

[3] `https://www.antlr.org`

```
...
Model:
    "declarations"
        declarations+=Declaration*
        fixedDeclarations+=FixedDeclaration+
    "rules"
        rules+=Rule*
    "cases"
        elements+=Case*
;

Declaration:
    name=ID "<->" path=STRING
;

FixedDeclaration:
    kind=Kind "<->" path=STRING
;

...

Rule:
    "def" name=ID "for" kind=Kind "{"
        "filter" decls+=[Declaration] ("||" decls+=[Declaration])*
        "such" "that" "size" compOp=(">"|"<"|"=="|"<="|">=") compValue=INT
    "}" |
    "def" "declare" name=ID
;

Case:
    "case" "for" kind=Kind condition=Expr "{"
        properties+=Property*
    "}"
;
...
```

Figure 4.2: Sample of DSL grammar in Xtext. Full grammar can be found in Appendix C.

```
...
override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
    ↪ IGeneratorContext context) {
    for (e:resource.allContents.toIterable.filter(Model)) {
        fsa.generateFile("generatedRequirements.xml",
            e.compile
        )
    }
}

def CharSequence compile(Model m)
'''
<model>
    «FOR r:m.rules»
        «r.compile»
    «ENDFOR»
    «FOR c:m.elements»
        «c.compile»
    «ENDFOR»
</model>
'''

def CharSequence compile(Rule r)
'''
«IF r.namet!="callMethodWithSameName"»
<declare rule="«r.name»" for="«getPath(r.kind)»"
    ↪ op="«opToString(r.compOp)»" size="«r.compValue»">
    «FOR d:r.decls»
        <filter instance="«d.path»"/>
    «ENDFOR»
</declare>
«ENDIF»
'''

def CharSequence compile(Case c)
'''
<case «c.kind»="«c.condition.compile»">
    «FOR p:c.properties»
        «p.compile»
    «ENDFOR»
</case>
'''
...
```

Figure 4.3: Sample of the code generator in Xtend. Full code generator can be found in Appendix D.

After completing the examination of the identifier implementation, the amount of found instances will be compared with the `comparisonValue` using the `comparisonOperator`. This leads to a boolean result, deciding whether the identifier with such implementation comply with this requirement or not. The full implementation of the rest of the analysis code can be found in the repository for the Java project[4].

[4] `https://git.app.uib.no/Emily.Nguyen/methodnameanalyser`

```java
public static boolean ruleChecker(Node caseTypeInput, String[]
    ↪ listOfInstances, String comparisonOperator, int comparisonValue) {
        List<Object> listOfFoundInstances = new ArrayList<Object>();
        for (String instanceString : listOfInstances) {
            List<Node> foundInstances = caseTypeInput.stream().filter(m
                ↪ -> {
                try {
                    return Class.forName(instanceString).isInstance(m);
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                    return false;
                }
            } )
            .collect(Collectors.toList());

            if (foundInstances.size() > 0) {
                listOfFoundInstances.add(foundInstances);
            }
        }

        boolean ruleResult = false;
        if (comparisonOperator.equals("LT")) {
            ruleResult = listOfFoundInstances.size() < comparisonValue;
        } else if (comparisonOperator.equals("GT")) {
            ruleResult = listOfFoundInstances.size() > comparisonValue;
        } else if (comparisonOperator.equals("EQ")) {
            ruleResult = listOfFoundInstances.size() == comparisonValue;
        } else if (comparisonOperator.equals("NE")) {
            ruleResult = listOfFoundInstances.size() != comparisonValue;
        } else if (comparisonOperator.equals("GE")) {
            ruleResult = listOfFoundInstances.size() >= comparisonValue;
        } else if (comparisonOperator.equals("LE")) {
            ruleResult = listOfFoundInstances.size() <= comparisonValue;
        }
    return ruleResult;
}
```

Figure 4.4: The Java method `ruleChecker`. Full implementation of analysis can be found in the repository for Java project.

Chapter 5

Evaluation: The Programmer's Lexicon

In this chapter, an evaluation of the analysing tool is presented with a case study, which is a lexicon of verbs outlined by Høst and Østvold [39] We will describe the implementation of a few entries of the lexicon; a complete implementation of the lexicon using our tool is presented in Appendix B.

5.1 Case Study

In this thesis, we consider the case study by Høst and Østvold, called *The Programmer's Lexicon, Volume I: The Verbs* [39]. It gives a list of the most commonly used method names, abstracted as verbs, by Java programmers. As we have already mentioned earlier, being able to create a name that can express the semantics of an implementation is crucial. Naming is abstract and the wrong name will lead to the wrong abstraction. Therefore, the mentioned case study [39] concludes that the problem of naming is significant for the task of programming, and formulates the research question: "Can we create a semantics which captures our common interpretation of method names?". Only the names of methods in Java are being investigated. The authors' approach is to encode the implementation of a method by its semantic properties, whether those properties are present or not. An example is to check if an implementation of a method performs any type checks, uses any local variables, throws exceptions, or so on.

In the study, the verbs in method names were analysed based on their actual usage [39]. Various verbs were determined by examining method implementations taken from a corpus of Java applications. Additionally, not all method names consist of a single verb, therefore most names needed to be abstracted to one verb only. For example, the method name "`checkAllSynonyms`" will be narrowed down to the verb "`check`", considering the verb is the action-oriented part of the full name. As a result, a domain-neutral lexicon is automatically generated involving only verbs, and represents the common practices among programmers.

5.2 The Lexicon

The Programmer's Lexicon is a collection of the most common names among Java programmers. It consist of two parts: a list of the verbs and a description to each verb, specifying the usage of the verb with how it is most commonly implemented. The following list is a sample of four of the entries from the lexicon [39]:

`create` Among the most common method names. Methods named `create` very often create objects. Furthermore, they rarely call methods of the same name, read state or contain loops.

`equals` Methods named `equals` never return void, throw exceptions, create objects, manipulate state or have no parameters. Furthermore, they very often call methods of the same name and perform type-checking. Finally, they often use local variables and read state. The name `equals` has a precise use.

`get` The most common method name. Methods named `get` often read state and have no parameters, and rarely return void, call methods of the same name, manipulate state, use local variables or contain loops. A similar name is `has`. Specialisations of `get` are `is` and `size`. A somewhat related name is `hash`.

`is` The third most common method name. Methods named `is` often have no parameters, and rarely return void, throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, perform type-checking or contain loops. The name `is` has a precise use. Generalisations of `is` are `has` and `get`. Somewhat related names are `accept`, `visit`, `hash` and `size`.

Phrases such as *always*, *very often*, *often*, *rarely*, *very seldom*, *never* are frequently used in the lexicon. As mentioned in Section 3.1.1, this is terminology defined in the case

study differentiating the quantile of the attributes. Table 3.4 gives the full list of all the phrases.

The lexicon can be converted to a list of rules on how method names with certain verbs should behave, considering this is how majority of Java programmers code. The authors of the case study analysed Java implementations, and created a lexicon of common usages.

In our thesis, we define a list of requirements for all the verbs in the lexicon, where these requirements can be used to analyse any Java source code, checking if it follows the common usages according to the lexicon. Chapter 3 explains how the requirements can be implemented, as well as how to analyse names from the source code. The implementation of the verb `create` is shown in Figure 5.1, and is specified with requirements according to the definition of `create` above.

```
case for method "create" {
    often
        createsObject
    rarely
        callsMethodWithSameName
    rarely
        readsState
    rarely
        containsLoop
}
```

Figure 5.1: Implementation of `create` in our DSL according to definition in lexicon.

Another example is the verb `is`, whose implementation is shown in Figure 5.2. This is a slightly longer case with over half of the requirements involving behaviours that rarely are present for a method named `is`.

The several rules, such as `createsObject`, `callsMethodWithSameName` and `performsTypeCheck`, are specified in Appendix A. The Appendix A also includes specifications of the needed declarations. Meanwhile, the implementation for rest of the verbs in the lexicon can be found in Appendix B.

To evaluate if the requirements have been implemented correctly and will behave as expected, a source code example will be utilised. Figure 5.3 is an example of a Java source code with four simple methods. Moreover, Figure 5.2 and Figure 5.1 is now modified to look like Figure 5.5 and Figure 5.4. Only the first line of each implemented

```
case for method "is" {
    rarely
        performsTypeCheck
    rarely
        throwsException
    rarely
        callsMethodWithSameName
    rarely
        usesLocalVariable
    rarely
        containsLoop
    rarely
        returnsVoid
    rarely
        throwsException
    rarely
        createsObject
    rarely
        manipulatesState
    often
        hasNoParameter
}
```

Figure 5.2: Implementation of `is` in our DSL according to definition in lexicon.

```
package com.github.example.name;
import java.util.ArrayList;
import java.util.Random;

public class ExampleClass implements IExample{
    public static void createLuckyNumber() {
        Random random = new Random();
        System.out.println("Lucky number: " + random.nextInt(100));
    }

    public static String greetWorld() {
        return "Hello World";
    }

    public static boolean isEven(int number) {
        return (number % 2 == 0);
    }

    public static ArrayList<String> makeList() {
        ArrayList<String> myList = new ArrayList<String>();
        return myList;
    }
}
```

Figure 5.3: Java source code example.

```
case for method #"create".JJ*.NN+ {
    often
        createsObject
    rarely
        callsMethodWithSameName
    rarely
        readsState
    rarely
        containsLoop
}
```

Figure 5.4: Modified implementation of `create` in our DSL.

```
case for method "is".("even"|"odd") {
    rarely
        performsTypeCheck
    rarely
        throwsException
    rarely
        callsMethodWithSameName
    rarely
        usesLocalVariable
    rarely
        containsLoop
    rarely
        returnsVoid
    rarely
        throwsException
    rarely
        createsObject
    rarely
        manipulatesState
    often
        hasNoParameter
}
```

Figure 5.5: Modified implementation of `is` in our DSL.

requirement has been changed to match the method names from the Java source code better, considering realistically not many method names consist of only a single verb.

Finally, following the explanation in Chapter 3.1.2 an analysis of Figure 5.3 will be executed with the implemented requirements. This will result in the following report as seen in Figure 5.6. The analysis report is printed out to the interactive command line.

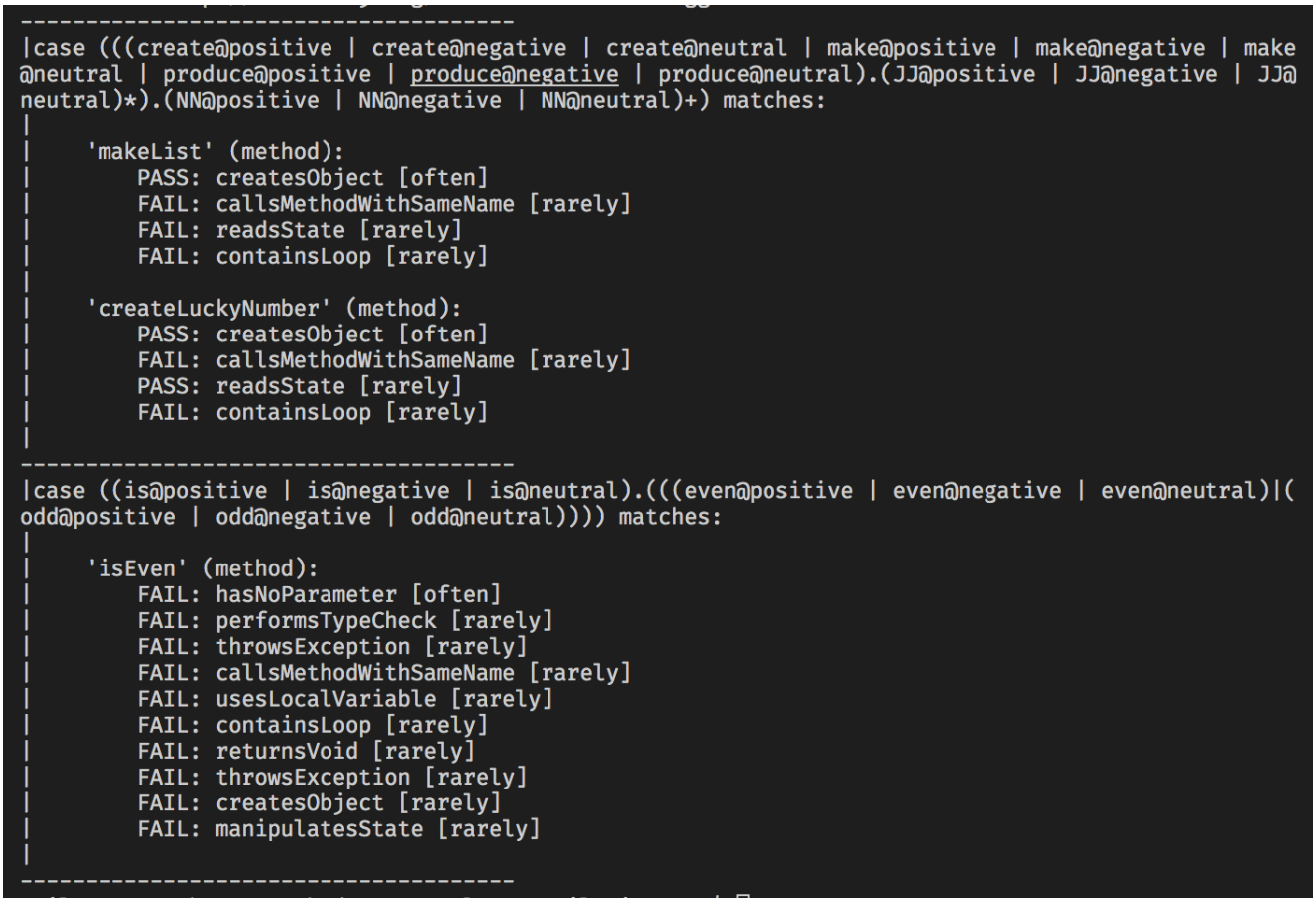

Figure 5.6: Interactive command line—analysis report.

First, it presents all the requirements as a case with its name condition. Then, all the identifier names that match any of the name conditions get listed under the case they match. After that, the body of each matched identifier name gets analysed with the requirements, and the results of what satisfy and what do not are listed underneath each method. The terms `PASS` and `FAIL` imply if the implementation of the method comply with the specific rule or not according to the frequency phrases they are supposed to follow such as *often*, *rarely* and *always*.

5.3 Retrospective

The lexicon from the case study *The Programmer's Lexicon, Volume I: The Verbs* [39] has been implemented, and the complete list can be found in Appendix B. The description of each verb has been encoded as the requirements when analysing the Java source code. Moreover, the resulting report of the analysis showcase not only that the system for analysing names works, but also shows several different approaches of linguistics integrated into the tool, which allows programmers to express a customised analysing system. This section has demonstrated that achieving the goal of this thesis, as defined in Section 1, is viable.

Chapter 6

Related Work

In this chapter, we present and discuss various of works related to the topic of this thesis. There will be two main focuses: naming in software languages and linguistics in software languages.

6.1 Naming in Software Languages

Several studies focus on names in software languages, be it research on how names and their implementation connects, or how to improve the art of naming code. Schäfer et al. presented the tool *J_L* that performs code refactoring on Java programs [73]. The implementation aims to enhance the naming and accessibility issues that are omnipresent in refactoring. To do this, Java gets translated to J_L with the help of accessibility constraints and reference constructions, allowing the refactoring to happen at the J_L level, and then it gets converted back to Java. Laitinen, on the other hand, researched on how to code better with names, and proposed methods and tools to assist natural naming usages in software programs [48]. Natural naming is building names that do not consist of abbreviations, and are believed to increase readability and comprehensibility of source code. Lawrie et al. partly agree with the approach from Laitinen, explaining that abbreviations do not carry a lot of information, thus is not as useful [50]. The paper presents a system of methods for translating and expanding abbreviated words into fuller names. However, not all abbreviations are less understandable than longer names [49]. Similarly, Deißenböck and Pizka studied the relations between names and concepts, leading to the support of concise names that reflect the implementation [21].

The authors claim that many programming languages accept arbitrary names, which can result in misleading names.

The work in this thesis focuses on the connection between names and concepts, like most of the mentioned works. Although it also analyses identifier names and the compliance of the corresponding implementation, its versatility for programmers to express a customised system for analysing names is different. The tool J_L performs code refactoring, aiming more towards the implementation of names in the code, and not on the semantics of names similarly to this work. When it comes to naming identifiers, one of the most studied topics is the use of abbreviations. In this work, programmers can decide for themselves to include or exclude abbreviations in names as preferred. Programmers also have the flexibility of specifying what types of names or sequences of words, should include or exclude what type of implementations. This work facilitates name consistency and accuracy.

6.2 Linguistics in Software Languages

Linguistics approaches have long been of interest for analysing software languages. *The Java Programmer's Phrase Book*[1] is compiled by Høst and Østvold, and captures the most commonly used grammatical structure and meaning of method names [38]. This book is automatically generated from studying a corpus of 100 open-source applications and libraries from different domains, containing at least one million methods. The work is built on the case study [39]—which we mentioned in details in Chapter 5—where it involves more than just the verbs from the case study. Additionally, several words can be combined to create phrases and result in new semantics of the methods. To do this, Høst and Østvold use NLP techniques such as part of speech tagging to unveil the structure of the method names. However, this is not the only work they have worked together using linguistics approaches [4].

The work in this thesis also involves a variety of word classes from part of speech tagging, such as nouns, verbs and adjectives. It is possible for programmers to combine and create customised conditions for names, and whether the semantics of the combined words will be the same or not as the semantics of the original words is up to each individual programmer, considering they can specify it as preferred. This work can cover the implementation for the *Java Programmer's Phrase Book* and more.

[1] `http://phrasebook.nr.no/phrasebook/index.html#`

Part of speech tagging seems to be one of the more used techniques in software language engineering. Kulkarni and Finlayson built *jMWE* which is a Java library involving part of speech tagging for testing and building detectors for multi-word expression (MWE) tokens [47]. Similarly, Butler et al. present the tool *NOMINAL* that utilise part of speech tagging as one of its inputs [13, 17]. NOMINAL is a library that checks naming conventions in Java, and allows specification of conventions. The study outcome was used to investigate the Java reference adherence to naming conventions. With all of so many studies involving part of speech, Olney et al. decided to investigate how accurate part of speech tagging is, and did tests on over 200 method names from open-source Java programs [65]. The outcome showed that customised project-designed part of speech taggers were significantly the most accurate taggers above others. Allamanis et al. also did research in how to suggest better names for programmers, where there were several NLP techniques utilised, especially language models (ML) and speech recognition, to support the analysis [4]. Several approaches from linguistics can be found the work of *NATURALIZE* framework as well [2] .

In the same way, the work in this thesis has implementations of linguistic approaches too. However, there are a few more approaches involved. As mentioned in Section 2.3.1, this work takes advantage of synonyms, part of speech tagging and sentiment analysis. NOMINAL resembles this work in how it allows specifications of conventions and integrates linguistics, but NOMINAL focuses the investigation on Java references, meanwhile this work analyses Java identifier names. Several studies and research implement approaches from linguistics to aid their work, likewise to this work, many aim to improve the usage software language such as identifier names.

Chapter 7

Conclusion and Future Work

Poor naming choices lead to a higher chance of poor code comprehension, however, choosing a concise and consistent name is challenging. This thesis has gathered and presented insight of how one can develop a system for analysing names in Java source code, helping programmers to choose and maintain better names. A DSL has been designed and implemented for programmers to easily specify requirements when analysing names. Programmers can define the declarations, rules and cases of the requirements, without having to worry about the underlying structure or details of the work. In addition, several approaches from linguistics have been implemented to improve the system for analysing names. Especially techniques from the subfield NLP of linguistics—such as synonyms, part of speech tagging and sentiment analysis—considering this work is a software analysis of natural languages. Linguistics supports and proves itself to be useful for analysing names in Java, as without linguistics the identifier names would not have been possible to be broken down and examined in the same way as they can now. Moreover, techniques from linguistics have made it possible for users to define requirements with customised names. Finally, an implementation of the lexicon from the case study *The Programmer's Lexicon, Volume I: The Verbs* has been implemented as a proof-of-concept [39]. This shows that the development of the system for analysing names behaves as intended. It is the same concept for other identifiers, such as classes and variables, although the implementation of the lexicon showcases only methods.

We have identified several interesting directions for further research, which are presented below.

Include more programming languages. The current system for analysing identifier names is only compatible for Java source code. This work uses the JavaParser library to parse through Java code, hence another parser is needed to include other programming languages. Examples of programming languages that can be implemented are: C#, COBOL and Python. Involving enough parsers can make this tool language-agnostic. This means programmers can specify requirements in the implemented DSL and analyse any source code regardless of programming language.

Supporting other natural languages. This involves extending the part of speech tagging for other languages, except English that is already implemented. Analysed names are separated into words and classified for what part of speech word class it is defined as. As mentioned in Section 2.3.1, each language is built differently, hence the analysis is done separately. To include more natural languages, the part of speech tagging for those languages need to be integrated to the system.

Extending the lexicon. The lexicon compiled by Høst and Østvold only involves verbs [39]s. There is a potential for extending the lexicon to involve more classes such as nouns, adjectives and adverbs. Differentiating sentiment loaded classes is also a possible addition to the lexicon. For example, negative adjectives could have other definitions than positive adjectives. Extending the lexicon this way can make it easier for programmers to create more precise and meaningful requirements.

Generating requirements from existing source code. Another possible direction is to design and build the tool to automatically generate requirements directly from an existing source code from the code base. Then, when the code base is extended with new source code, this new code can be checked against the extracted requirements.

Integrating with IDEs and code repositories. Integrating the name analysis tool with existing IDEs (for instance, by implementing the tools as plugins for Eclipse-, Visual Studio-, or IntelliJ-based IDEs) will improve accessibility and user-friendliness Bringing this idea further, one can imagine integrating name analysis tools with cloud-based code repositories hosting systems, such as GitHub[1] or GitLab[2]. Such integration would enable the name analysis feature in the IDE or code repository, making it more accessible and easier to utilise, due to not needing several systems running to perform the name analysis.

[1] `https://github.com`
[2] `https://gitlab.com`

Supporting more linguistics-based approaches. There is potential for including more linguistic approaches to the tool. Examples of other linguistic techniques are *semantic role labelling* (SRL) [70], hyponyms and hypernyms, *named entity recognition* (NER) [56] and *word-sense disambiguation* (WSD) [61]. Most of these approaches stem from NLP, which are relevant for further development of this tool. SRL can identify sentences, breaking it down to smaller elements to classify and label the elements for which semantic role it holds. Hyponyms and hypernyms have been explained in Section 2.3.1. NER can determine proper names and categorise names accordingly, whether it is a name for people, location or brand etc. WSD filters and selects the meaning that makes the most sense for a word, considering most words have several meanings, thus causes ambiguity.

Bibliography

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: an effective verification process. *IEEE software*, 6(3):31–36, 1989.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635883.
URL: `https://doi.org/10.1145/2635868.2635883`.

[3] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[5] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016.

[6] Mehdi Assefi, Guangchi Liu, Mike P Wittie, and Clemente Izurieta. An experimental evaluation of apple siri and google speech recognition. *Proccedings of the 2015 ISCA SEDE*, 118, 2015.

[7] Aliaksandr Autayeu. extjwnl (extended java wordnet library). `http://extjwnl.sourceforge.net`, last viewed 10.05.2022.

[8] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese, Jan Köhnlein, Knut Wannheden, and Sebastian Zarnekow. Xtext user guide. *Dostupné*

z WWW: http://www. eclipse. org/Xtext/documentation/1_0_1/xtext. html, page 7, 2008.

[9] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[10] Douglas Biber, Susan Conrad, and Randi Reppen. *Corpus linguistics: Investigating language structure and use*. Cambridge University Press, 1998.

[11] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445, 2009.

[12] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983. ISSN 0020-7373. doi: https://doi.org/10.1016/S0020-7373(83)80031-5.
URL: `https://www.sciencedirect.com/science/article/pii/S0020737383800315`.

[13] Simon Butler. *Analysing Java Identifier Names*. Open University (United Kingdom), 2016.

[14] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35, 2009. doi: 10.1109/WCRE.2009.50.

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

[16] Simon Butler, Michel Wermelinger, and Yijun Yu. Investigating naming convention adherence in java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50. IEEE, 2015.

[17] Simon Butler, Michel Wermelinger, and Yijun Yu. Investigating naming convention adherence in java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50. IEEE, 2015.

[18] Simon Butler, Michel Wermelinger, and Yijun Yu. A survey of the forms of java reference names. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 196–206. IEEE, 2015.

[19] KR1442 Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pages 603–649, 2020.

[20] Emilio Collar Jr and Ricardo Valerdi. Role of software readability on software development cost. Technical report, 2006.

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

[22] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[23] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[24] Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. How developers choose names. *IEEE Transactions on Software Engineering*, 48(1):37–52, 2022. doi: 10.1109/TSE.2020.2976920.

[25] Ronen Feldman. Techniques and applications for sentiment analysis. *Communications of the ACM*, 56(4):82–89, 2013.

[26] Brian Fitzgerald. The transformation of open source software. *MIS quarterly*, pages 587–598, 2006.

[27] Eclipse Foundation. About the eclipse foundation. `https://www.eclipse.org/org/`, last viewed 30.05.2022, .

[28] Eclipse Foundation. Xtext - web editor support. `https://www.eclipse.org/Xtext/documentation/330_web_support.html`, last viewed 27.05.2022, .

[29] The Apache Software Foundation. The apache software foundation. `https://www.apache.org`, last viewed 30.05.2022, .

[30] The Apache Software Foundation. Apache opennlp. `https://opennlp.apache.org`, last viewed 30.05.2022, .

[31] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[32] Angelo Gargantini and Marco Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on*

Software Testing, Verification and Validation Workshops (ICSTW), pages 308–317. IEEE, 2018.

[33] David Geer. Eclipse becomes the dominant java ide. *Computer*, 38(7):16–18, 2005.

[34] Diana Gray. Why does java remain so popular? `https://blogs.oracle.com/oracleuniversity/post/why-does-java-remain-so-popular`, last viewed 30.02.2022.

[35] Remo Gresta, Vinicius Durelli, and Elder Cirilo. Naming practices in java projects: An empirical study. In *XX Brazilian Symposium on Software Quality*, pages 1–10, 2021.

[36] Felienne Hermans. *The Programmer's Brain: What every programmer needs to know about cognition*. Simon and Schuster, 2021.

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

[38] Einar W Høst and Bjarte M Østvold. The java programmer's phrase book. In *International Conference on Software Language Engineering*, pages 322–341. Springer, 2008.

[39] Einar W Høst and Bjarte M Østvold. The programmer's lexicon, volume i: The verbs. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202. IEEE, 2007.

[40] Sun Microsystems Inc. Fortran 77 4.0 reference manual. `https://wwwcdf.pd.infn.it/localdoc/f77_sun.pdf`, last viewed 27.05.2022.

[41] JetBrains. Java - programming. the state of developer ecosystem in 2021 infographic. `https://www.jetbrains.com/lp/devecosystem-2021/java/`, last viewed 27.05.2022.

[42] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Professional, 2003.

[43] Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 444–463, 2010.

[44] Veton Kepuska and Gamal Bohouta. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*, pages 99–103. IEEE, 2018.

[45] Chanwoo Kim, Sungsoo Kim, Kwangyoun Kim, Mehul Kumar, Jiyeon Kim, Kyungmin Lee, Changwoo Han, Abhinav Garg, Eunhyang Kim, Minkyoo Shin, et al. End-to-end training of a large vocabulary end-to-end speech recognition system. In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 562–569. IEEE, 2019.

[46] Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel. Code conventions for the java programming language. *Sun Microsystems*, 1999.

[47] Nidhi Kulkarni and Mark Finlayson. jmwe: A java toolkit for detecting multi-word expressions. In *Proceedings of the Workshop on Multiword Expressions: from Parsing and Generation to the Real World*, pages 122–124, 2011.

[48] Kari Laitinen. *Natural naming in software development and maintenance*. Citeseer, 1995.

[49] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

[50] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 213–222, 2007. doi: 10.1109/SCAM.2007.17.

[51] Sheen S Levine and Michael J Prietula. Open collaboration for innovation: Principles and performance. *Organization Science*, 25(5):1414–1433, 2014.

[52] John Lyons. *Introduction to theoretical linguistics*, volume 510. Cambridge university press, 1968.

[53] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[54] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.

[55] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. Addison-Wesley Professional, 2010.

[56] Andrei Mikheev, Marc Moens, and Claire Grover. Named entity recognition without gazetteers. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 1–8, 1999.

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

[58] Ruslan Mitkov. *The Oxford handbook of computational linguistics*. Oxford University Press, 2004.

[59] Bob Muglia. Bob muglia: Visual studio 2010 global launch keynote. `https://news.microsoft.com/2010/04/12/bob-muglia-visual-studio-2010-global-launch-keynote/`, last viewed 30.02.2022.

[60] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 2011.

[61] Roberto Navigli. Word sense disambiguation: A survey. *ACM computing surveys (CSUR)*, 41(2):1–69, 2009.

[62] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[63] Kunio Nishiyama. Adjectives and the copulas in japanese. *Journal of East Asian Linguistics*, 8(3):183–222, 1999.

[64] Linguistic Society of America. The science of linguistics. `https://www.linguisticsociety.org/resource/science-linguistics`, last viewed 30.05.2022.

[65] Wyatt Olney, Emily Hill, Chris Thurber, and Bezalem Lemma. Part of speech tagging java method names. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 483–487. IEEE, 2016.

[66] Oracle. Developing java applications. https://docs.oracle.com/javase/7/docs/technotes/guides/, last viewed 27.05.2022, .

[67] Oracle. Sun java enterprise system user management guide. https://www.oracle.com/corporate/pressrelease/oracle-buys-sun-042009.html, last viewed 30.05.2022, .

[68] Oracle. Timeline of key java milestones. https://www.oracle.com/java/moved-by-java/timeline/, last viewed 30.05.2022, .

[69] Oracle. Coding conventions for the java programming language: 9. naming convention. https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html, last viewed 31.05.2022, .

[70] Martha Palmer, Daniel Gildea, and Nianwen Xue. Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, 3(1):1–103, 2010.

[71] Martin Richards. Bcpl: A tool for compiler writing and system programming. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 557–566, 1969.

[72] Norbert Schmitt. *An introduction to applied linguistics*. Routledge, 2013.

[73] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A comprehensive approach to naming and accessibility in refactoring java programs. *IEEE Transactions on Software Engineering*, 38(6):1233–1257, 2012. doi: 10.1109/TSE.2012.13.

[74] Charles Simonyi. *Meta-programming: a software production method.* Stanford University, 1977.

[75] Charles Simonyi. Hungarian notation. *MSDN Library, November*, 1999.

[76] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[77] Larry Squire, Darwin Berg, Floyd E Bloom, Sascha Du Lac, Anirvan Ghosh, and Nicholas C Spitzer. *Fundamental neuroscience*. Academic press, 2012.

[78] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[79] Gérard Valenduc and Patricia Vendramin. Digitalisation, between disruption and evolution. *Transfer: European Review of Labour and Research*, 23(2):121–134, 2017.

[80] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[81] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8–style guide for python code. *Python. org*, 1565, 2001.

[82] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[83] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 16(3), 2010.

[84] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and GH Wachsmuth. Dsl engineering-designing, implementing and using domain-specific languages. 2013.

[85] Henry George Widdowson. *Linguistics*. Oxford University Press, 1996.

[86] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697–709, 1986.

[87] Fengyi Zhang, Bihuan Chen, Rongfan Li, and Xin Peng. A hybrid code representation learning approach for predicting method names. *Journal of Systems and Software*, 180:111011, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111011.
URL: `https://www.sciencedirect.com/science/article/pii/S0164121221001084`.

Appendix A

Declarations and Rules Implementation

Here is the implementation of the sections *declarations* and *rules* in our DSL. There are a few custom specifications and pre-defined types included. More details about the specification can be found in subsection 3.1.1. Appendix B has the implementation based on the declarations and rules specified here.

Type	**Section**	**Implementation in our DSL**
Parameter	Declaration	`Parameter <-> "com.github.javaparser.ast.body .Parameter"`
Variable declarator	Declaration	`VariableDeclarator <-> "com.github.javaparser. ast.body.VariableDeclarator"`
Object creation expression	Declaration	`ObjectCreationExpr <-> "com.github.javaparser .ast.expr.ObjectCreationExpr"`
Instance of expression	Declaration	`InstanceOfExpr <-> "com.github.javaparser.ast .expr.InstanceOfExpr"`

Cast expression	Declaration	`CastExpr <-> "com.github.javaparser.ast.expr .CastExpr"`
Field access expression	Declaration	`FieldAccessExpr <->"com.github.javaparser.ast .expr.FieldAccessExpr"`
Assign expression	Declaration	`AssignExpr <-> "com.github.javaparser.ast.expr .AssignExpr"`
Variable declaration expression	Declaration	`VariableDeclarationExpr <-> "com.github .javaparser.ast.expr.VariableDeclarationExpr"`
Void type	Declaration	`VoidType <-> "com.github.javaparser.ast.type .VoidType"`
While statement	Declaration	`WhileStmt <-> "com.github.javaparser.ast.stmt .WhileStmt"`
For each statement	Declaration	`ForEachStmt <-> "com.github.javaparser.ast.stmt .ForEachStmt"`

For statement	Declaration	`ForStmt <-> "com.github.javaparser.ast.stmt .ForStmt"`
Do statement	Declaration	`DoStmt <-> "com.github.javaparser.ast.stmt .DoStmt"`
Throw statement	Declaration	`ThrowStmt <-> "com.github.javaparser.ast.stmt .ThrowStmt"`
Package	Declaration pre-defined	`package <-> "com.github.javaparser.ast .PackageDeclaration"`
Class	Declaration pre-defined	`class <-> "com.github.javaparser.ast.body .ClassOrInterfaceDeclaration"`
Interface	Declaration pre-defined	`interface <-> "com.github.javaparser.ast.body .ClassOrInterfaceDeclaration"`
Variable	Declaration pre-defined	`variable <-> "com.github.javaparser.ast.body .VariableDeclarator"`

Method	Declaration pre-defined	`variable <-> "com.github.javaparser.ast.body .MethodDeclaration"`

Method has no parameter	Rule	`def hasNoParameter for method {`
		`  filter Parameter`
		`  such that size < 1`
		`}`

Method returns void	Rule	`def returnsVoid for method {`
		`  filter VoidType`
		`  such that size == 1`
		`}`

Method throws Exception	Rule	`def throwsException for method {`
		`  filter ThrowStmt`
		`  such that size > 0`
		`}`

Method uses local variable	Rule	`def usesLocalVariable for method {`
		`  filter VariableDeclarator`
		`  such that size > 0`
		`}`

Method contains loop	Rule	`def containsLoop for method {`
		`  filter WhileStmt || ForEachStmt || ForStmt`
		`  || DoStmt`
		`  such that size > 0`
		`}`

Method creates object	Rule	`def createsObject for method {`
		`  filter ObjectCreationExpr`
		`  such that size > 0`
		`}`

Method performs type check	Rule	`def performsTypeCheck for method {`
		`  filter InstanceOfExpr || CastExpr`
		`  such that size > 0`
		`}`

Method reads state	Rule	`def readsState for method {`
		`  filter FieldAccessExpr`
		`  such that size > 0`
		`}`

Method manipulates state	Rule	`def manipulatesState for method {`
		`  filter AssignExpr || VariableDeclarationExpr`
		`  such that size > 0`
		`}`

Method calls method with same name	Rule pre-defined	`def declare callsMethodWithSameName`

Appendix B

Lexicon With Implementation

Here is the implementation of each verb from *The Lexicon* [39] in our DSL. The implementation is based on the description of the verb. Furthermore, this implementation is also considered the last section of the requirement specifications, where declarations and rules from the first two sections showed in Appendix A can be found in this implementation. More details about the specification can be found in subsection 3.1.1.

Verb	Description from lexicon [39]	Implementation in our DSL

accept	*Methods named "accept" very seldom read state. Furthermore, they rarely throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, have no parameters, perform type-checking or contain loops. The name "accept" has a precise use. A similar name is "visit". Generalisations of "accept" are handle and "initialize". Somewhat related names are "set", "end", "is" and "insert".*	`case for method "accept" {`
		`  very-seldom`
		`  readsState`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  manipulatesState`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  hasNoParameter`
		`  rarely`
		`  performsTypeCheck`
		`  rarely`
		`  containsLoop`
		`}`

`action`	*Methods named "action" never call methods of the same name. Furthermore, they very often read state. Finally, they often return void, and rarely throw exceptions, have no parameters or contain loops. The name "action" has a precise use. Similar names are "remove" and "add".*	`case for method "action" {`
		`  very-often`
		`  readsState`
		`  rarely`
		`  throwsException`
		`  never`
		`  callsMethodWithSameName`
		`  rarely`
		`  hasNoParameter`
		`  rarely`
		`  containsLoop`
		`}`

`add`	*Among the most common method names. Methods named "add" often read state. Similar names are "remove" and "action".*	`case for method "add" {`
		`  often`
		`  readsState`
		`}`

`check`	*Methods named "check" very often throw exceptions. Furthermore, they often create objects and contain loops, and rarely call methods of the same name. Unfortunately, "check" is an imprecise name for a method.*	`case for method "check" {`
		`  very-often`
		`  throwsException`
		`  rarely`
		`  callsMethodWithSameName`
		`  often`
		`  createsObject`
		`  often`
		`  containsLoop`
		`}`

`clear`	*Methods named "clear" very often have no parameters. Furthermore, they often return void, call methods of the same name and manipulate state, and rarely create objects, use local variables or perform type-checking. A generalisation of "clear" is "reset". A somewhat related name is "close".*	`case for method "clear" {`
		`  often`
		`  callsMethodWithSameName`
		`  rarely`
		`  createsObject`
		`  often`
		`  manipulatesState`
		`  rarely`
		`  usesLocalVariable`
		`  very-often`
		`  hasNoParameter`
		`  rarely`
		`  performsTypeCheck`
		`  often`
		`  returnsVoid`
		`}`

`close`	*Methods named "close" often return void, call methods of the same name, manipulate state, read state and have no parameters, and rarely create objects or perform type-checking. A generalisation of "close" is "validate". A somewhat related name is "clear".*	`case for method "close" {`
		`  often`
		`  returnsVoid`
		`  often`
		`  callsMethodWithSameName`
		`  often`
		`  manipulatesState`
		`  often`
		`  readsState`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  performsTypeCheck`
		`}`

`create`	*Among the most common method names. Methods named "create" very often create objects. Furthermore, they rarely call methods of the same name, read state or contain loops.*	`case for method "create" {`
		`  often`
		`  createsObject`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  readsState`
		`  rarely`
		`  containsLoop`
		`}`

`do`	*Methods named "do" often throw exceptions and perform type-checking, and rarely call methods of the same name. Unfortunately, "do" is an imprecise name for a method.*	`case for method "do" {`
		`  often`
		`  throwsException`
		`  often`
		`  performsTypeCheck`
		`  rarely`
		`  callsMethodWithSameName`
		`}`

`dump`	*Methods named "dump" never throw exceptions. Furthermore, they very often create objects and use local variables, and very seldom read state. Finally, they often call methods of the same name and contain loops, and rarely manipulate state. The name "dump" has a precise use.*	`case for method "dump" {`
		`  very-often`
		`  createsObject`
		`  very-often`
		`  usesLocalVariable`
		`  often`
		`  callsMethodWithSameName`
		`  often`
		`  containsLoop`
		`  never`
		`  throwsException`
		`  very-seldom`
		`  readsState`
		`  rarely`
		`  manipulatesState`
		`}`

`end`	*Methods named "end" often return void, and rarely create objects, use local variables, read state or contain loops. Generalisations of "end" are "handle" and "initialize". A specialisation of "end" is "insert". Somewhat related names are "accept", "set", "visit" and "write".*	`case for method "end" {`
		`  often`
		`  returnsVoid`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  readsState`
		`  rarely`
		`  containsLoop`
		`}`

`equals`	*Methods named "equals" never return void, throw exceptions, create objects, manipulate state or have no parameters. Furthermore, they very often call methods of the same name and perform type-checking. Finally, they often use local variables and read state. The name "equals" has a precise use.*	`case for method "equals" {`
		`  very-often`
		`  performsTypeCheck`
		`  very-often`
		`  callsMethodWithSameName`
		`  often`
		`  usesLocalVariable`
		`  often`
		`  readsState`
		`  never`
		`  returnsVoid`
		`  never`
		`  throwsException`
		`  never`
		`  createsObject`
		`  never`
		`  manipulatesState`
		`  never`
		`  hasNoParameter`
		`}`

`find`	*Methods named "find" very often use local variables and contain loops. Furthermore, they often perform type-checking, and rarely return void.*	`case for method "find" {`
		`  very-often`
		`  usesLocalVariable`
		`  very-often`
		`  containsLoop`
		`  often`
		`  performsTypeCheck`
		`  rarely`
		`  returnsVoid`
		`}`

`generate`	*Methods named "generate" often create objects, use local variables and contain loops, and rarely call methods of the same name. Unfortunately, "generate" is an imprecise name for a method.*	`case for method "generate" {`
		`  often`
		`  createsObject`
		`  often`
		`  usesLocalVariable`
		`  often`
		`  containsLoop`
		`  rarely`
		`  callsMethodWithSameName`
		`}`

`get`	*The most common method name. Methods named "get" often read state and have no parameters, and rarely return void, call methods of the same name, manipulate state, use local variables or contain loops. A similar name is "has". Specialisations of "get" are "is" and "size". A somewhat related name is "hash".*	`case for method "get" {`
		`  often`
		`  readsState`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  returnsVoid`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  manipulatesState`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  containsLoop`
		`}`

`handle`	*Methods named "handle" often read state, and rarely call methods of the same name. A similar name is "initialize". Specialisations of "handle" are "accept", "set", "visit", "end" and "insert".*	`case for method "handle" {`
		`  often`
		`  readsState`
		`  rarely`
		`  callsMethodWithSameName`
		`}`

`has`	*Methods named "has" often have no parameters, and rarely return void, throw exceptions, create objects, manipulate state, use local variables or perform type-checking. The name "has" has a precise use. A similar name is "get". "Specialisations" of "has" are "is" and "size". A somewhat related name is "hash".*	`case for method "has" {`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  returnsVoid`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  manipulatesState`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  performsTypeCheck`
		`}`

`hash`	*Methods named "hash" always have no parameters, and never return void, throw exceptions, create objects or perform type-checking. Furthermore, they very often call methods of the same name. Finally, they often read state, and rarely manipulate state or use local variables. The name "hash" has a precise use. Somewhat related names are "has", "is", "get" and "size".*	`case for method "hash" {`
		`  always`
		`  hasNoParameter`
		`  often`
		`  readsState`
		`  never`
		`  returnsVoid`
		`  never`
		`  throwsException`
		`  never`
		`  createsObject`
		`  never`
		`  performsTypeCheck`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  usesLocalVariable`
		`}`

`init`	*Methods named "init" very often manipulate state. Furthermore, they often return void, create objects and have no parameters, and rarely call methods of the same name.*	`case for method "init" {`
		`  very-often`
		`  manipulatesState`
		`  often`
		`  returnsVoid`
		`  often`
		`  createsObject`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  callsMethodWithSameName`
		`}`

`initialize`	*Methods named "initialize" often return void and manipulate state, and rarely call methods of the same name or read state. A similar name is "handle". Specialisations of "initialize" are "accept", "set", "visit", "end" and "insert".*	`case for method "initialize" {`
		`  often`
		`  returnsVoid`
		`  often`
		`  manipulatesState`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  readsState`
		`}`

`insert`	*Methods named "insert" often throw exceptions, and rarely create objects, read state, have no parameters or contain loops. Generalisations of "insert" are "handle", "end" and "initialize". Somewhat related names are "accept", "set", "visit" and "write".*	`case for method "insert" {`
		`  often`
		`  throwsException`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  readsState`
		`  rarely`
		`  hasNoParameter`
		`  rarely`
		`  containsLoop`
		`}`

`is`	*The third most common method name. Methods named "is" often have no parameters, and rarely return void, throw exceptions, call methods of the same name, create objects, manipulate state, use local variables, perform type- checking or contain loops. The name "is" has a precise use. Generalisations of "is" are "has" and "get". Somewhat related names are "accept", "visit", "hash" and "size".*	`case for method "is" {`
		`  rarely`
		`  performsTypeCheck`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  containsLoop`
		`  rarely`
		`  returnsVoid`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  manipulatesState`
		`  often`
		`  hasNoParameter`
		`}`

`load`	*Methods named "load" very often use local variables. Furthermore, they often throw exceptions, create objects, manipulate state, perform type-checking and contain loops. Unfortunately, "load" is an imprecise name for a method.*	`case for method "load" {`
		`  very-often`
		`  usesLocalVariable`
		`  often`
		`  throwsException`
		`  often`
		`  createsObject`
		`  often`
		`  manipulatesState`
		`  often`
		`  performsTypeCheck`
		`  often`
		`  containsLoop`
		`}`

`make`	*Methods named "make" very often create objects. Furthermore, they rarely return void, throw exceptions, call methods of the same name or contain loops.*	`case for method "make" {`
		`  very-often`
		`  createsObject`
		`  rarely`
		`  returnsVoid`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  containsLoop`
		`}`

`new`	*Methods named "new" never contain loops. Furthermore, they very seldom use local variables. Finally, they often call methods of the same name and create objects, and rarely return void, manipulate state or read state.*	`case for method "new" {`
		`  often`
		`  callsMethodWithSameName`
		`  very-seldom`
		`  usesLocalVariable`
		`  rarely`
		`  readsState`
		`  rarely`
		`  returnsVoid`
		`  often`
		`  createsObject`
		`  rarely`
		`  manipulatesState`
		`  never`
		`  containsLoop`
		`}`

`next`	*Methods named "next" very often manipulate state and read state. Furthermore, they often throw exceptions and have no parameters, and rarely return void.*	`case for method "next" {`
		`  very-often`
		`  manipulatesState`
		`  very-often`
		`  readsState`
		`  often`
		`  throwsException`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  returnsVoid`
		`}`

`parse`	*Among the most common method names. Methods named "parse" very often call methods of the same name, read state and perform type-checking. Furthermore, they rarely use local variables. The name "parse" has a precise use.*	`case for method "parse" {`
		`  very-often`
		`  performsTypeCheck`
		`  very-often`
		`  callsMethodWithSameName`
		`  very-often`
		`  readsState`
		`  rarely`
		`  usesLocalVariable`
		`}`

`print`	*Methods named print often call methods of the same name and contain loops, and rarely throw exceptions or manipulate state.*	`case for method "print" {`
		`  very-often`
		`  callsMethodWithSameName`
		`  often`
		`  containsLoop`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  manipulatesState`
		`}`

process	*Methods named "process" very often use local variables and contain loops. Furthermore, they often throw exceptions, create objects, read state and perform type-checking, and rarely call methods of the same name. Unfortunately, "process" is an imprecise name for a method.*	`case for method "process" {`
		`  often`
		`  performsTypeCheck`
		`  rarely`
		`  callsMethodWithSameName`
		`  often`
		`  usesLocalVariable`
		`  very-often`
		`  containsLoop`
		`  often`
		`  throwsException`
		`  often`
		`  createsObject`
		`  often`
		`  readsState`
		`  often`
		`  performsTypeCheck`
		`}`

`read`	*Methods named "read" often throw exceptions, call methods of the same name, create objects, manipulate state, use local variables and contain loops. Unfortunately, "read" is an imprecise name for a method.*	`case for method "read" {`
		`  often`
		`  throwsException`
		`  often`
		`  callsMethodWithSameName`
		`  often`
		`  createsObject`
		`  often`
		`  manipulatesState`
		`  often`
		`  usesLocalVariable`
		`  often`
		`  containsLoop`
		`}`

`remove`	*Among the most common method names. Methods named "remove" often throw exceptions. Similar names are "add" and "action".*	`case for method "remove" {`
		`  often`
		`  throwsException`
		`}`

`reset`	*Methods named "reset" very often manipulate state. Furthermore, they often return void and have no parameters, and rarely create objects, use local variables or perform type-checking. A specialisation of "reset" is "clear".*	`case for method "reset" {`
		`  very-often`
		`  manipulatesState`
		`  often`
		`  returnsVoid`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  performsTypeCheck`
		`}`

`run`	*Among the most common method names. Methods named "run" very often read state. Furthermore, they often have no parameters, and rarely call methods of the same name.*	`case for method "run" {`
		`  very-often`
		`  readsState`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  callsMethodWithSameName`
		`}`

`set`	*The second most common method name. Methods named "set" very often manipulate state, and very seldom use local variables or read state. Furthermore, they often return void, and rarely call methods of the same name, create objects, have no parameters, perform type-checking or contain loops. The name "set" has a precise use. Generalisations of "set" are "handle" and "initialize". Somewhat related names are "accept", "visit", "end" and "insert".*	`case for method "set" {`
		`  very-often`
		`  manipulatesState`
		`  often`
		`  returnsVoid`
		`  very-seldom`
		`  usesLocalVariable`
		`  very-seldom`
		`  readsState`
		`  rarely`
		`  callsMethodWithSameName`
		`  rarely`
		`  createsObject`
		`  rarely`
		`  hasNoParameter`
		`  rarely`
		`  performsTypeCheck`
		`  rarely`
		`  containsLoop`
		`}`

`size`	*Methods named "size" always have no parameters, and never return void, create objects, manipulate state, perform type-checking or contain loops. Furthermore, they very seldom use local variables. Finally, they rarely read state. The name "size" has a precise use. Generalisations of "size" are "has" and "get". Somewhat related names are "is" and "hash".*	`case for method "size" {`
		`  always`
		`  hasNoParameter`
		`  never`
		`  returnsVoid`
		`  never`
		`  createsObject`
		`  never`
		`  manipulatesState`
		`  never`
		`  performsTypeCheck`
		`  never`
		`  containsLoop`
		`  very-seldom`
		`  usesLocalVariable`
		`  rarely`
		`  readsState`
		`}`

`start`	*Methods named "start" often return void, manipulate state and read state.*	`case for method "start" {`
		`  often`
		`  returnsVoid`
		`  often`
		`  manipulatesState`
		`  often`
		`  readsState`
		`}`

`to`	*Among the most common method names. Methods named "to" very often call methods of the same name and create objects. Furthermore, they often have no parameters, and rarely return void, throw exceptions, manipulate state or perform type-checking.*	`case for method "to" {`
		`  very-often`
		`  callsMethodWithSameName`
		`  rarely`
		`  returnsVoid`
		`  very-often`
		`  createsObject`
		`  rarely`
		`  throwsException`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  manipulatesState`
		`  rarely`
		`  performsTypeCheck`
		`}`

`update`	*Methods named "update" often return void and read state.*	`case for method "update" {`
		`  often`
		`  returnsVoid`
		`  often`
		`  readsState`
		`}`

`validate`	*Methods named "validate" very often throw exceptions. Furthermore, they often create objects and have no parameters, and rarely manipulate state. A specialisation of "validate" is "close".*	`case for method "validate" {`
		`  very-often`
		`  throwsException`
		`  often`
		`  createsObject`
		`  often`
		`  hasNoParameter`
		`  rarely`
		`  manipulatesState`
		`}`

`visit`	*Methods named "visit" rarely throw exceptions, use local variables, read state or have no parameters. A similar name is "accept". Generalisations of "visit" are "handle" and "initialize". Somewhat related names are "set", "end", "is" and "insert".*	`case for method "visit" {`
		`  rarely`
		`  throwsException`
		`  rarely`
		`  usesLocalVariable`
		`  rarely`
		`  readsState`
		`  rarely`
		`  hasNoParameter`
		`}`

`write`	*Among the most common method names. Methods named "write" often return void and call methods of the same name, and rarely have no parameters. Somewhat related names are "end" and "insert".*	`case for method "write" {`
		`  often`
		`  returnsVoid`
		`  often`
		`  callsMethodWithSameName`
		`  rarely`
		`  hasNoParameter`
		`}`

Appendix C

Grammar for the Implemented DSL

This is the grammar in Xtext for the implemented DSL, and can be found in the repository that hosts the Xtext project[1] in the file `MethodName.xtext`[2].

```
grammar org.xtext.example.methodname.MethodName with
    ↪ org.eclipse.xtext.common.Terminals

generate methodName "http://www.xtext.org/example/methodname/MethodName"

Model:
    "declarations"
        declarations+=Declaration*
        fixedDeclarations+=FixedDeclaration+
    "rules"
        rules+=Rule*
    "cases"
        elements+=Case*
;

Declaration:
    name=ID "<->" path=STRING
;

FixedDeclaration:
    kind=Kind "<->" path=STRING
;

enum Kind:
    KIND_INTERFACE="interface"|
    KIND_PACKAGE="package" |
    KIND_CLASS="class" |
    KIND_METHOD="method" |
    KIND_VAR="variable"
;

Rule:
    "def" name=ID "for" kind=Kind "{"
        "filter" decls+=[Declaration] ("||" decls+=[Declaration])*
        "such" "that" "size" compOp=(">"|"<"|"=="|"<="|">=") compValue=INT
    "}" |
    "def" "declare" name=ID
;
```

[1] https://git.app.uib.no/Emily.Nguyen/nameanalyser

[2] https://git.app.uib.no/Emily.Nguyen/nameanalyser/-/blob/master/org.xtext.example.methodname/src/org/xtext/example/methodname/MethodName.xtext

```
38
39 Case:
40     "case" "for" kind=Kind condition=Expr "{"
41         properties+=Property*
42     "}"
43 ;
44
45 Expr: Or;
46
47 Or returns Expr:
48     And ({Or.left=current} "|" right=And)*;
49
50 And returns Expr:
51     Primary ({And.left=current} concatKind=("."|"...") right=Primary)*;
52
53 Primary returns Expr:
54     GroupExpr |
55     Atomic
56 ;
57
58 GroupExpr:
59     hasNot?="!"? "(" expr=Expr ")" card=CardinalityModifier?
60 ;
61
62 Atomic returns Expr:
63     {StringConst}
64         hasNot?="!"? value=STRING card=CardinalityModifier? ("[" sentiment=Sentiment
               ↪ "]")? |
65     {POSValue}
66         hasNot?="!"? pos=POS card=CardinalityModifier? ("[" sentiment=Sentiment "]")? |
67     {SynonymConst}
68         hasNot?="!"? "#" value=STRING card=CardinalityModifier? ("["
               ↪ sentiment=Sentiment "]")?
69 ;
70
71 enum Sentiment:
72     SENT_UNSPECIFIED |
73     SENT_NEUTRAL="neutral" |
74     SENT_POSITIVE="positive" |
75     SENT_NEGATIVE="negative"
76 ;
77
78 enum POS:
79     POS_CC="CC" |
80     POS_CD="CD" |
81     POS_DT="DT" |
82     POS_EX="EX" |
83     POS_FW="FW" |
84     POS_IN="IN" |
85     POS_JJ="JJ" |
86     POS_JJR="JJR" |
87     POS_JJS="JJS" |
88     POS_LS="LS" |
89     POS_MD="MD" |
90     POS_NN="NN" |
91     POS_NNS="NNS" |
92     POS_NNP="NNP" |
93     POS_NNPS="NNPS" |
94     POS_PDT="PDT" |
95     POS_POS="POS" |
96     POS_PRP="PRP" |
97     POS_RB="RB" |
98     POS_RBR="RBR" |
99     POS_RBS="RBS" |
100    POS_RP="RP" |
101    POS_SYM="SYM" |
102    POS_TO="TO" |
103    POS_UH="UH" |
```

```
    POS_VB="VB" |
    POS_VBD="VBD" |
    POS_VBG="VBG" |
    POS_VBN="VBN" |
    POS_VBP="VBP" |
    POS_VBZ="VBZ" |
    POS_WDT="WDT" |
    POS_WP="WP" |
    POS_WRB="WRB"
;

enum CardinalityModifier:
    CARD_1 |
    CARD_0_N="*" |
    CARD_1_N="+" |
    CARD_0_1="?"
;

Property:
    frequency=Frequency? criterion=[Rule];

enum Frequency:
    ALWAYS="always" |
    VERYOFTEN="very-often" |
    OFTE="often" |
    RARELY="rarely" |
    VERYSELDOM="very-seldom" |
    NEVER="never"
;
```

Appendix D

Code Generator for the Implemented DSL

This is the code generator in Xtend for the implemented DSL, and can be found in the repository that hosts the Xtext project[1] in the file `MethodNameGenerator.xtend`[2].

```
1  package org.xtext.example.methodname.generator
2
3  import org.eclipse.emf.ecore.resource.Resource
4  import org.eclipse.xtext.generator.AbstractGenerator
5  import org.eclipse.xtext.generator.IFileSystemAccess2
6  import org.eclipse.xtext.generator.IGeneratorContext
7
8  import org.xtext.example.methodname.methodName.Model
9  import org.xtext.example.methodname.methodName.Case
10 import org.xtext.example.methodname.methodName.Expr
11 import org.xtext.example.methodname.methodName.Property
12 import org.xtext.example.methodname.methodName.And
13 import org.xtext.example.methodname.methodName.Or
14 import org.xtext.example.methodname.methodName.StringConst
15 import org.xtext.example.methodname.methodName.POSValue
16 import org.xtext.example.methodname.methodName.GroupExpr
17 import org.xtext.example.methodname.methodName.SynonymConst
18 import org.xtext.example.methodname.methodName.Rule
19 import org.xtext.example.methodname.methodName.Kind
20
21 class MethodNameGenerator extends AbstractGenerator {
22
23     def getPath(Kind pathKind) {
24         return pathKind
25     }
26
27     def operatorToString(String operator) {
28         switch operator {
29             case ">":"GT"
30             case "<":"LT"
31             case "==":"EQ"
32             case "<=":"LE"
33             case ">=":"GE"
34             case "!=":"NE"
35         }
36     }
37
```

[1] https://git.app.uib.no/Emily.Nguyen/nameanalyser

[2] https://git.app.uib.no/Emily.Nguyen/nameanalyser/-/blob/master/org.xtext.example.methodname/src/org/xtext/example/methodname/generator/MethodNameGenerator.xtend

```
38    override void doGenerate(Resource resource, IFileSystemAccess2 fsa,
          ↪ IGeneratorContext context) {
39        for (e:resource.allContents.toIterable.filter(Model)) {
40            fsa.generateFile("generatedRequirements.xml",
41                e.compile
42            )
43        }
44    }
45
46    def CharSequence compile(Model m)
47    '''
48    <model>
49        «FOR r:m.rules»
50            «r.compile»
51        «ENDFOR»
52        «FOR c:m.elements»
53            «c.compile»
54        «ENDFOR»
55    </model>
56    '''
57
58    def CharSequence compile(Rule r)
59    '''
60    «IF r.name!="callMethodWithSameName"»
61    <declare rule="«r.name»" for="«getPath(r.kind)»"
          ↪ op="«operatorToString(r.compOp)»" size="«r.compValue»">
62        «FOR d:r.decls»
63            <filter instance="«d.path»"/>
64        «ENDFOR»
65    </declare>
66    «ENDIF»
67    '''
68
69    def CharSequence compile(Case c)
70    '''
71    <case «c.kind»="«c.condition.compile»">
72        «FOR p:c.properties»
73            «p.compile»
74        «ENDFOR»
75    </case>
76    '''
77
78    def dispatch CharSequence compile(Expr e) {
79        switch e {
80            Or: {'''(«e.left.compile»|«e.right.compile»)'''}
81            And: {'''(«e.left.compile».«e.right.compile»)'''}
82            GroupExpr: {'''(«e.expr.compile»«IF
                  ↪ e.card!=e.card.CARD_1»«e.card»«ENDIF»)'''}
83            StringConst: {'''(«e.value»«IF
                  ↪ e.sentiment==e.sentiment.SENT_UNSPECIFIED»@positive | «
                  ↪ e.value»@negative | «
                  ↪ e.value»@neutral«ELSE»@«e.sentiment»«ENDIF»)«IF
                  ↪ e.card!=e.card.CARD_1»«e.card»«ENDIF»'''}
84            POSValue: {'''(«e.pos»«IF
                  ↪ e.sentiment==e.sentiment.SENT_UNSPECIFIED»@positive | «
                  ↪ e.pos»@negative | «e.pos»@neutral«ELSE»@«e.sentiment»«ENDIF»)«IF
                  ↪ e.card!=e.card.CARD_1»«e.card»«ENDIF»'''}
85            SynonymConst: {'''«IF e.sentiment==
                  ↪ e.sentiment.SENT_UNSPECIFIED»(«Synonym.generateAllSynonyms(e.value,
                  ↪ "unspecified")»«ELSEIF e.sentiment==
                  ↪ e.sentiment.SENT_NEUTRAL»(«Synonym.generateAllSynonyms(e.value,
                  ↪ e.sentiment.SENT_NEUTRAL.toString)»«ELSEIF e.sentiment==
                  ↪ e.sentiment.SENT_POSITIVE»(«Synonym.generateAllSynonyms(e.value,
                  ↪ e.sentiment.SENT_POSITIVE.toString)»«ELSEIF e.sentiment==
                  ↪ e.sentiment.SENT_NEGATIVE»(«Synonym.generateAllSynonyms(e.value,
                  ↪ e.sentiment.SENT_NEGATIVE.toString)»«ENDIF»)«IF
                  ↪ e.card!=e.card.CARD_1»«e.card»«ENDIF»'''}
86        }
```

```
87      }
88
89      def CharSequence compile(Property p)
90      '''
91      <«p.frequency»><«p.criterion.name»/></«p.frequency»>
92      '''
93  }
```