Comprehensible Method Names: Focusing on the Nouns

Dennis van Leeuwen

July 23, 2012

Master Software Engineering

Supervisor : dr. Jurgen Vinju

Publication status: public

University of Amsterdam

Contents

Abstract **4**

Preface **5**

1 Introduction **6**

1.1 Motivation 6

1.2 Focusing on the nouns in method names 7

1.3 Defining co-occurrence of nouns 7

1.4 Research question 9

1.5 Scope 9

1.6 Comprehensible method names 10

1.7 Organization of the thesis 11

2 Background and Related Work **12**

2.1 Consistently named identifiers 12

2.2 Assessment of the verb in the method name 13

2.3 Keyword extraction from documents 14

3 Research method **15**

3.1 Mining the strength of the co-occurrence of nouns 15

3.2 Manual verification of the comprehensibility 18
3.3 Corpus 19

4 The strength of the co-occurrence of nouns 21
4.1 Introduction 21
4.2 Research method 21
4.3 Results 23
4.4 Discussion 25
4.5 Conclusion 25

5 Comprehensibility of methods 27
5.1 Introduction 27
5.2 Research method 27
5.3 Results 29
5.4 Discussion 33
5.5 Conclusion 34

6 Relating the co-occurrence of nouns to the size of a method 36
6.1 Introduction 36
6.2 Research method 37
6.3 Results 37
6.4 Discussion 38
6.5 Conclusion 38

7 Using the verbs to find the co-occurrences of nouns 39
7.1 Introduction 39
7.2 Research method 40
7.3 Results 43

7.4 Discussion 51
7.5 Conclusion 52

8 Conclusion **53**

9 Future work **55**

References **55**

A Example to demonstrate initial approach **58**

B Analyzed methods on comprehensibility **61**
B.1 Methods with co-occurrence of nouns 61
B.2 Methods without co-occurrence of nouns 64

C Found lexicon entries **68**
C.1 Methods starting with get 68
C.2 Methods starting with set 68
C.3 Methods starting with is 69
C.4 Methods starting with create 69
C.5 Methods starting with add 69

D Comparison of lexicon entries **70**
D.1 Methods starting with get 70
D.2 Methods starting with set 70
D.3 Methods starting with is 71
D.4 Methods starting with create 71
D.5 Methods starting with add 72

Abstract

The maintenance phase is the most costly phase in the software development cycle. During maintenance the software is studied in order to create enough understanding to be able to comprehend the source code. This comprehension process can be improved if comprehensible identifiers are used.
In this this is focused on finding hard-to-comprehend methods using the co-occurrence of nouns. A co-occurrence of nouns is present in a method if any of the nouns in the method name occurs at least once in the identifier information in the method implementation.
The co-occurrence of nouns was researched using a corpus containing a total of 31 open source systems. The co-occurrence of nouns was found to occur in 83% of the methods containing at least one noun in the method name and one noun in the identifiers in the implementation.
If a co-occurrence was present in a method, the chance this method was also found to be comprehensible was higher. This suggests that using the co-occurrence of nouns hard-to-comprehend methods can be found. However, the results suggest that not all the methods without the co-occurrence of nouns are hard-to-comprehend. Therefore the co-occurrence of nouns is useful to find methods that need further manual inspection.
Using the statements and expressions in the implementation of a method, we searched for the noun causing the presence of a co-occurrence of nouns. The co-occurring noun was found consistently in specific identifiers used in specific statement and expressions. Therefore not only a possible way to assess the comprehensibility of the method was found, but also a possible way to find an appropriate noun to use in the method name was found.

Preface

This thesis is the result of the master project done between the March 7, 2012 and July 21, 2012. The master project is the final fulfilment in order to obtain the degree: Master Of Science. The master project was performed at CWI, in Dutch Centrum Wiskunde en Informatica. The global idea of assessing method names was based on the master thesis subject of Jouke Stoel.

Since my idea was based on Jouke Stoel's master project, I would like to thank Jouke Stoel for (unconsciously) inspiring me for a great master project topic. Furthermore, I would like to thank Jurgen Vinju and Luuk Stevens for their valuable feedback during the writing of this thesis.

Chapter 1

Introduction

1.1 Motivation

The larger a system becomes, the harder it will be to maintain it. During maintenance it is therefore preferred to use source code that is as easy as possible to read, so the maintainer is able to understand and modify the source code as fast as possible. Typically, maintenance consumes over 70% of the total life cycle cost of a software product[2]. During that time the source code will be studied in order to create enough understanding to be able to comprehend the source code.
Sametinger stated that the comprehension process takes over 50% of the time spent on the maintenance task[13]. Therefore, if the comprehensibility can be improved, the time and cost consumed during maintenance can be reduced[3]. Fang stated that hardly any software is maintained for its whole life by the original author[5]. It is therefore important that the source code is easy to comprehend, since the programmer himself probably will not be the only one maintaining the source code.

The operations in source code can be explained in two possible ways:

- Use comments to explain the work flow in the program.
- Use meaningful identifiers to explain the elements in the work flow and the actions that are executed.

Lawrie et al. confirmed that the quality of identifiers effects the comprehensibility of the program[11]. To avoid programmers naming the identifiers in a

hard-to-comprehend way, we argue that there should be an automatic way to assess whether the method name is accurately explaining its implementation. This will assist the programmer to improve the quality of identifiers.

1.2 Focusing on the nouns in method names

We focus on method names, since comprehensible identifiers are needed and methods are one kind of identifiers. Furthermore, methods names are interesting, since we assume they are a description of the method implementation. The assumption is that method names represent the answer on the question: "**What** does the method do?"
The relations that we think exist between method name and method implementation are shown in the table 1.1.

	Action	Direct or Indirect object
Method name	Verb	Noun
Method implementation	Expressions & Statements	Types & Objects

Table 1.1: Relation between the name and the implementation of a method.

The behavior is described by an action, which is reflected by a verb in the method name and statements and expressions in the implementation. The object used(if it is used) by the action is reflected as a direct or indirect object in the method name. We suspect this object is also reflected by a noun in at least one identifier in the method implementation.
The work[8, 16] of Høst and Østvold and Stoel suggests programmers consistently use a specific verb in the method name if certain statements and expressions are used in the method implementation. Since research has been done on using an appropriate verb in the method name, we focus on the use of an appropriate noun in the method name. We check if an appropriate noun is used by researching the relation between the nouns used in the method name and the nouns used in the implementation of the method.

1.3 Defining co-occurrence of nouns

We focus on researching the relation between the nouns used in the method name and the nouns used in the implementation of the method. We will refer to this relation as the co-occurrence of nouns. A co-occurrence of nouns is

present if any of the nouns in the method name is equal to any of the nouns used in the identifier information in the implementation of the method. A co-occurrence of nouns is present in the following methods in example 1.1:

- "getLicensePlateNumber", since the implementation contains the nouns: "license", "plate" and "number" in a variable name.
- "createCar", since the base type of the created object contains the noun "car".

```
1  public abstract class Car{
2    private String licensePlateNumber;
3
4    public Car(String licensePlateNumber){
5      this.licensePlateNumber = licensePlateNumber;
6    }
7
8    public String getLicensePlateNumber(){
9      return this.licensePlateNumber;
10   }
11 }
12
13 public class Ferrari extends Car{
14   public Ferrari(String licensePlateNumber){
15     super(licensePlateNumber);
16   }
17 }
18
19 public class MainProgram{
20   public static void main(String args[]){
21     Car myFerrari = createCar("NL-23-07");
22     System.out.println(myFerrari.getLicensePlateNumber());
23   }
24   public static Car createCar(String licensePlateNumber){
25     return new Ferrari(licensePlateNumber);
26   }
27 }
```

Figure 1.1: Example to show the co-occurrence of nouns in methods

To indicate the usefulness of the co-occurrence of nouns, the strength of the co-occurrence of nouns is used. The strength of a co-occurrence of nouns is calculated with the following formula:

$$strength = \frac{allMethodsWithCoOccurringNoun}{allMethodsWithNouns} * 100\%$$

'allMethodsWithCoOccurringNoun' is the amount of methods with the presence of a co-occurrence of nouns. 'allMethodsWithNouns' is the amount of methods with at least one noun in the method name and at least one noun in the identifier information in the implementation of the method.

1.4 Research question

Our hypothesis is that a co-occurrence of nouns is present in almost any method, since method names represent their implementation. By assessing the comprehensibility of the method and relating this comprehensibility to the presence of a co-occurrence of nouns, we search for a relation in the source code to find hard-to-comprehend methods. The main research question discussed in this thesis is:

Can the co-occurrence of nouns be used to find hard-to-comprehend methods?

To answer this question, the following subquestions are researched:

1. What is the strength of the co-occurrence of nouns?
2. What is the relation between the co-occurrence of nouns and the comprehensibility of the method?
3. What is the relation between the co-occurrence of nouns and the size of a method?
4. Can the verb-noun relation in the method implementation be used to find the noun occurring in the method name?

1.5 Scope

We focus on analyzing the nouns in both method name and in the identifiers in the implementation of the method. However, nouns can occur in different identifiers in the source code. In this thesis we focus on any identifier except packages, since we think that the nouns available in a package name are imprecise. We explain this using the example in figure 1.2.

```
1 public interface Adult{}
2 public class HugeAdult implements Adult{}
3
4 public class MainProgram{
5   void Adult createPerson(){
6     return new persons.adults.HugeAdult();
7   }
8 }
```

Figure 1.2: Example to show why we do not analyze package name

We argue this method is not totally comprehensible, since the noun 'person' is too abstract in the scope of this method. The name of the method does not explain the implementation of the method accurately, since an adult is created instead of a person. Although there exists a co-occurrence of nouns in the package name and the method name, we argue that the package names are too general.

1.5.1 Identifier Information

We have established an exhaustive list of properties of identifiers. A co-occurrence of nouns can only exist between a noun in the method name and a noun in a property listed below.

- Type: Variables and methods have a type. The type of a method is its return type. A type can be an interface, class or enumeration.
- Name: Variable and methods have a name. Types have a name as well, but that name is equal to the Type itself. Names of types are therefore ignored, since it is useless to process a word twice.
- Base Type: Types can be derived of other types: base types. Since all classes used in Java are derived from java.util.Object, we ignore this type as base type. The type java.util.Object is not ignore if it is used as a normal type.

Since we compare nouns to find a co-occurrence of nouns, we will give a definition of the equality of nouns. Our definition is:

Nouns are equal if and only if the stemmed form of both nouns is equal.

1.6 Comprehensible method names

Assessing the comprehensibility of method names is hard, since comprehensibility is something subjective[4]. Any person can have a different opinion about a method name. The Java Code Convention states that names should be short and meaningful[1]. Names should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. However, assessing

this is impossible, since we cannot retrieve how the name was designed to indicate this if it was not documented.
To reduce the subjective judgment of comprehensibility as much as possible, we defined criteria. A comprehensible method should apply to these criteria to be qualified as comprehensible. These criteria were found in earlier work[4, 9]:
First of all, a method name should be appropriate. It is appropriate if the method name faithfully describes the named entity and is not misleading[9]. Use of homonyms and synonyms can be misleading, therefore only one term should be used in method name and body for a single concept[4].
Secondly, a method name should be meaningful. Meaningful was defined as "a programmer can to some extent understand a program entity from its name"[9]. This means an implementation of a method can be hidden to a programmer causing the programmer to know the methods behavior by only depending on its name.

1.7 Organization of the thesis

In chapter 2 the background information and related work is discussed. This work is discussed since it is used as basement for our analysis to assess the comprehensibility of methods. Chapter 3 explains the research approach.
Chapters 4, 5, 6 and 7 are used to discuss the subquestions of this research. Each chapter is used for a single subquestion. A conclusion of the work is given in chapter 8. Furthermore, an overview of future work is given in chapter 9.
Results related to our research are found in the appendices. These results have been found by us, but are not that important to be shown in the previous chapters.

Chapter 2

Background and Related Work

In this chapter we identify main points in earlier research. We describe their work so it can be used as basis for the assessment of good identifier names. We furthermore give some background information about the terms used in this thesis.

2.1 Consistently named identifiers

Deißenböck and Pizka found that identifiers are a major part of the source code[4]. Since identifiers are adjustable, they can have an effect on the comprehensibility of the source code. Deißenböck and Pizka mention three possible causes of having not properly named identifiers. These causes are based on findings in a numerous amount of code bases:

1. Identifiers can be arbitrarily chosen by developers.
2. Developers have only limited knowledge about the names already used somewhere else in the system.
3. Identifiers are subject to decay during system evolution. The concepts they refer to are altered or abandoned without properly adapting the names. One reason for this is the lack of tool support for globally renaming sets of identifiers referring to the same concept.

To point out the use of multiple words for the same concept, a the tool called IDD(IDentifier Dictionary) was developed. The tool is described as:

"The concept of the IDD is inspired and works similar to a Data Dictionary. Basically, it is a database that stores information about all identifiers such as their name, the type of the object being identified and a comprehensive description."

Using only one word for a single concept allows the programmer to understand a basic vocabulary of words, such that all concepts can be understood. The consistent usage of words for concepts would improve the comprehensibility of the identifiers, although the usefulness in practice has not been researched yet. We therefore assess the consistent usage of a nouns since our assumption is that the consistency is relevant.

2.2 Assessment of the verb in the method name

Høst and Østvold related the action described in the method name to the implementation of the method. They found that programmers consistently use specific verbs in the method name if specific combinations of traceable attributes were present in the implementation of a method[7]. Using this consistent relation, a list of lexicon entries was constructed with the verbs programmers use in the method name and traceable patterns present in the method implementation with this verb[7]. Høst and Østvold used a total of 10 traceable attributes in their initial work[7]. They extended the set of traceable attributes in later work[8, 9].

Furthermore a tool, called Lancelot[9], was developed to aid programmers in finding and fixing naming bugs[9] in methods. Lancelot suggests method names by using a set of traceable attributes and the phrase of the method name mined from a corpus. This phrase includes the verb and the structure of the method name. Although the value of Lancelot has not been research yet in practice, it is good to point out inconsistencies in the source code. These findings can be good for further (manual) analysis.

2.2.1 Extended set of traceable attributes called nano-patterns

Although Høst and Østvold introduced an initial set of 10 traceable attributes, Singer et al. extended this catalogue of traceable attributes to a total of 17. Singer et al. called them fundamental nano-patterns[14]. The major difference with the traceable attributes of Høst and Østvold is that

Singer et al. detect writing to local variable and reading of local variable instead of just using of local variable, like Høst and Østvold do. Furthermore, Singer et al. uses extra patterns to discriminate the method implementations.

2.2.2 Verifying the usefulness of nano-patterns to assess the verb

Stoel verified the usefulness of the nano-patterns in a method implementation to check for naming anomalies in the method name[16]. The results of this research suggest that using this assessment of method names is useful. However, this approach was only used to assess the 'action-token'. If a verb occurs as the first word in a method name, it is an action-token. The results suggest that methods found using his approach have a higher chance to contain naming anomalies than randomly picked methods.

2.3 Keyword extraction from documents

Matsuo and Ishizuka researched how relevant keywords can be extracted from a document using the information available in the document itself. Comparable research topics like "automatic term recognition" in the context of computational linguistics and "automatic indexing" or "automatic keyword extraction" can be found in the research areas of information retrieval[12]. Matsuo and Ishizuka first extracted the frequent occurring terms. Next, the "co-occurrences" of a term and frequent terms were counted. If a term appears frequently with a particular subset of terms, the term is likely to have important meaning.

Keyword extraction algorithms can be useful in the context of finding the most relevant noun in the method name. The keywords are represented by the words in the method name. The terms appearing in the document are comparable to the words used in the identifier information in the method implementation. Using a similar approach might help us in properly assessing the relevance of the nouns in the method name.

Chapter 3

Research method

In this chapter the global setup of the research is discussed. The research can be divided in two parts.
First, the strength of the co-occurrence of nouns is mined from a corpus. This step is done completely automatic using a custom made mining tool.
The second step is the manual assessment of the comprehensibility of two samples taken from the corpus. Using the samples the usefulness of the found relations and their strength can be validated. In this chapter we only discuss the global setup. In each chapter for each subquestion we explain the research method in more detail and how we use it specifically for that research question.
In the sections below we discuss each global part of the research method in detail.

3.1 Mining the strength of the co-occurrence of nouns

The first step of our approach is to extract information about the co-occurrence of nouns. This is done to explore the co-occurrence of nouns in general and should produce and motive the knowledge needed to start designing a method for the detection of hard-to-comprehend methods. In the sections below each step in the mining process of information about the co-occurrence of the nouns is discussed.

3.1.1 Extracting data

The first step is extracting all the necessary data from the source code. We use a corpus to extract data from, since we want to analyze the co-occurrence of nouns in all kinds of applications. This corpus is explained in detail in section 3.3. To collect the data from the corpus we use a domain specific language for source code analysis and manipulation called Rascal [10]. We extract all the method names and the identifier information(See section 1.5.1) from the implementation of the method using Rascal.

3.1.2 Filtering data

When the methods and the profiles are extracted, they are filtered to avoid noise in the results. The filtering process is done according to the guidelines stated in the Java Code Convention. We have listed all of filtering steps below.
We filter:

- Methods if the method or class of the method contains a $-sign. We argue that identifiers that contain $-signs are usually generated.
- Methods containing $-signs in the identifiers in the implementation, since these methods can be generated as well.
- Methods using an underscore. The Java Code Convention states that methods should be composed in a camelCase style. We argue that underscores should be avoided since programmers are recommended to use a different way to separate the words in a method name.
- Abstract methods or methods without implementation are useless for analysis and are therefore filtered. Methods having only parameters do not count as methods with an implementation.
- Constructors. The names of constructors are always equal to their class name. Since we do not focus on comprehensible class names, we ignore constructors.
- Methods of anonymous classes. We argue that objects of anonymous classes are practically always used to override a certain method and they are also used only once. Since the class name cannot be created by the programmer, the name is not relevant.

3.1.3 Preparing data

The third step is preparing the data for analysis. The collected data is transformed so the detection algorithm of a co-occurrence of nouns only needs to detect the noun relation. Each of the step executed in the preparing data phase are discussed below.

Splitting/Decomposition

The splitting phase splits each of the identifier names into an array of words. The splitting algorithm is equal to the decomposition algorithm used by Høst and Østvold[8]. It splits each name in different words based on its camelCase form. The Java Code Convention states Java programmers should use camelCase to name identifiers. Furthermore, we split names based on the occurrences of underscores, since the Java Naming Conventions recommends to name constants using an underscore separator.

Part Of Speech Tagging

Tagging is the identification of the lexical form of a word. The tagging algorithm of Høst and Østvold is used to tag words in a method name. This algorithm is estimated to be 97% accurate[6] and is based on the words available in WordNet 3.0[15]. The tagging process of words in identifier names other than method names is developed by us, but is also based on the algorithm of Høst and Østvold. It was necessary to make a small change to the original algorithm of Høst and Østvold, since the tagging process of words in identifiers other than method names should not get a preference based on the grammatical structure of methods.

Part Of Speech Stemming

After the identification of the lexical form of each word, the nouns need to be transformed into their stemmed form. This is necessary to be able to compare the nouns on equality. For example, 'alumni' would not be matched to 'alumnus' using their normal form, but since we use the stemmed form, both nouns are equal after stemming. The stemming algorithm is based on WordNet 3.0.

3.1.4 Detecting the co-occurrence of nouns

When the data is prepared for analysis, the last step is to detect the co-occurrence of nouns. This detection algorithm differs for each research question and is therefore discuss in detail in each chapter specific for each research question.

3.2 Manual verification of the comprehensibility

The second step is verifying the usefulness of the strength found in the automatic mining step. A co-occurrence of nouns can be measured to be strong, but if the relation does not tell you something about the comprehensibility, the relation is useless to find less comprehensible methods. We therefore manually assessed two randomly taken samples from the corpus. One sample contained methods with a co-occurrence of nouns and one sample contained methods without a co-occurrence of nouns. By comparing the comprehensibility and the amount of meaningless and inappropriate names found in both samples, we assessed whether one of the two samples was more comprehensible.
The comprehensibility of each method was analyzed by only looking at the method name, class name of the method and the method implementation. We have chosen to analyze the comprehensibility this way, since method names should be abstractions of their implementation. Our opinion is that a good method name hides it implementation and should only show its intent in the name. A programmer that needs to use the methods should be able to use the methods without having to read the implementation of the method.

3.2.1 Threat to validity

The comprehensibility of the methods was assessed by ourselves. If any confusion was noticed assessing the comprehensibility of the method we have ask for a second opinion from a fellow researcher. This was done to reduce the influence of a bias, since we might have unconsciously graded a method as comprehensible or hard-to-comprehend, since a presence or absence of a co-occurrence of nouns can be easily spotted in some methods. We have furthermore selected a number of criteria to assess the method names. Since these criteria are more concrete than just the question "Does the method name fit to the method implementation?", we tried to reduce the influence

of a bias. However, a bias can still have influence on our assessment and it is therefore a threat to validity.

3.3 Corpus

Our research depends on what Java programmers do. A way to analyze what they do, is by analyzing lots of source code and detect consistencies in the source code. We analyzed Java source code from a total of 31 systems with a total of over 130,000 methods, excluding filtered methods. The source code of all the systems is publicly available. Since Rascal can only analyze source code from successfully compile projects and it was hard to compile all the projects in a full corpus, we took a subset of the corpuses of both Høst and Østvold and Singer et al. instead of a full corpus.
We have selected a number of 9 categories. Each category is represented proportionally by acquire systems with a total of 15,000 methods, excluding filtered methods, per category. This was possible to do for all the categories except 'Jakarta common utilities'. This category contains only about 8,000 methods, since we were not able to setup more projects from this category. An overview of the corpus is shown in table 3.1.

Category	Name	Version
Benchmarking	Jikes RVM	2.9.1
Desktop application	JEdit	4.3
	JHotDraw	709
	ArgoUML	0.24
Jakarta common utilities	Commons Collections	3.2
	Commons IO	1.3.1
	Commons Codec	1.3
	Commons Lang	2.3
	Commons Digester	1.8
	Commons Net	1.4.1
	Commons HttpClient	1.0.1
Language and language tools	JRuby	0.9.2
	Antlr	2.7.6
	ASM	2.2.3
	Bcel	5.2
	BSF	2.4.0
	Polyglot	2.1.0
Middleware, frameworks and toolkits	Spring	2.0.2
	TranQL	1.3
	Tapestry	4.0.2
Programmer tools	FitNesse	2011-01-04
	JUnit	4.2
	Ant	1.7.0
	Velocity	1.4
Server and database	JBoss	3.2.7
Utilities and libraries	JarJar Links	0.7
	Ognl	2.6.9
	Hibernate	3.2.1
	XML Batik	1.6
XML tools	Xalan-J	2.7.0
	Castor	1.1

Table 3.1: Overview of the total corpus.

Chapter 4

The strength of the co-occurrence of nouns

4.1 Introduction

We expect the direct or indirect object used by the action in the method to be reflected as a noun in the method name and as noun in the types or objects used in the method implementation(See section 1.2). We assume a co-occurrence of nouns is present very often and thus the strength of the co-occurrence of nouns to be high.

The research question discussed in this chapter is:
What is the strength of the co-occurrence of nouns?

4.2 Research method

The approach used in this research question is almost completely described in section 3.1. The co-occurrence of nouns is detected by searching for a noun in the method name equal to any of the nouns found in the identifier information in the method implementation. This is including the parameters of the method.

4.2.1 Initial approach

In our first attempt to measure the strength of the relation we measured how strong the nouns in the method name are related to the nouns used in the identifier information in the implementation of the method.
We measured the strength by counting the amount of times a noun occurred in the implementation. The amount of times the noun was counted compared to the amount of times other nouns were counted would suggest how important the noun was. We expected the most important noun to occur in the method name.
However, this way of measurement is tricky, since a programmer is allowed to use temporary variables and use a certain noun, that does not occur in the method name, in these variables as much as he likes. To determine if co-occurrence is present, we have to set a threshold of a minimum amount of occurrences. However, this can be relative to the amount of different nouns present in the method. Furthermore, we do not know whether the co-occurrence of a noun is something that is valuable at all. Therefore we decided to focus on only the co-occurrence without relating this to the times the noun needs to be counted to be relevant. The decision not to count the times of occurrences results in a more simple measurement, since this is a binary decision and therefore no thresholds needed to be defined.
An example to motivate this decision is shown in appendix A.

4.2.2 Final approach

In the final approach we measured in how many methods the co-occurrence of nouns was present. We first checked if a noun occurred in the method name and occurred again in the identifier information in the implementation. Using this approach, we measured if the implementation is at least someway related to the object that is found in the method name.

To measure the strength of the co-occurrence of nouns we have first analyzed each noun on each position in the method name. We measure how strong the co-occurrence between each noun in the method name and the nouns in the identifier information from the implementation is by taking a collection of methods. This is done to explore if any of the nouns in the method name has a more powerful role in the method name. We measure the strength of a co-occurrence of nouns position based and per noun. Furthermore, the strength of a co-occurrence of nouns is also measured for any noun in the

method name. Using both measurements we check if both approaches are both as useful to find a co-occurrence of nouns.

4.3 Results

4.3.1 Position specific nouns in a method name

We first measure the strength of the co-occurrence of nouns based on the position of the nouns. The results of the experiment are shown in table 4.1 and 4.2.

Position	# Methods	# Methods with co-occurrence	Strength (in %)
1	98843	76735	77.63
2	44294	34590	78.09
3	10507	8250	78.52
4	2002	1464	73.13
5	433	306	70.67
6	134	78	58.21
7	59	25	42.37
8	36	13	36.11
9	22	13	59.09
10	9	2	22.22
11	2	1	50.00
12	1	0	0.00

Table 4.1: Strength of the co-occurrence of the noun found on a position based on the beginning of the method

The first line tells us that there were 98843 methods with at least one noun in the method name and at least one noun in the method implementation. The first noun in these method names was used in 76735 methods in the identifier information in the implementation of the method. This means that in 77.63% of the methods a co-occurrence of nouns was found.

In the second experiment the position of each noun was based on the last noun in the method name. On position 'last' the last occurrence of a noun is used to measure the strength of the co-occurrence of nouns. The results of this experiment are shown in table 4.2

Position	# Methods	# Methods with co-occurrence	Strength (in %)
11 before last	1	0	0.00
10 before last	2	0	0.00
9 before last	9	2	22.22
8 before last	22	15	68.18
7 before last	36	19	52.78
6 before last	59	23	38.98
5 before last	134	71	52.99
4 before last	433	249	57.51
3 before last	2002	1307	65.28
2 before last	10507	7690	73.19
1 before last	44294	33986	76.73
last	98843	78115	79.03

Table 4.2: Strength of the co-occurrence of the noun found on a posititon based on the end of the method.

4.3.2 All the nouns in a method name

Another experiment not focusing on a noun on a fixed position, but on all the nouns in the method name was done. The results of this experiment are shown in table 4.3.

	Name with noun(s)	Name without noun(s)	Total
Implementation with noun(s)	98843	27604	126447
Implementation without noun(s)	2882	1238	4120
Total	101725	28842	130567

Table 4.3: Distribution of methods.

Focusing only on the methods with at least one noun in the name and at least one noun in the identifier information in the implementation, we searched for methods with and without co-occurrences of nouns. The results of this experiment are shown in table 4.4

	# Methods	% of total
Methods with co-occurrence	82767	83.74
Methods without co-occurrence	16076	16.26
Total	98843	100.00

Table 4.4: Distribution of methods with a noun in the name and a noun in the implementation.

4.4 Discussion

We have researched the relation between each noun on each position in the method name. We think it is remarkable that programmers create methods with 10 or more nouns. Although, this research is not about the assessment of using the most concise method name as possible, we argue it is good to investigate these methods in more detail in future work. Furthermore, the results show that the last noun has the strongest co-occurrence. However, the results suggest that the last noun is not always used in the implementation and nouns on different positions can occur in the implementation in those methods.

In 2882 of the 101014 methods a noun was found in the name, but no noun was found in the implementation. These methods might contain not meaningful or inappropriate identifiers, but this depends whether there were undetectable 'objects' used in the method body. We only filtered methods with a completely empty body. These methods can still return a primitive or a string object for example. If this is done without using an identifier, no noun can be detected. Therefore the detection of the co-occurrence of nouns should be extended to be sufficient to detect a co-occurrence of noun in more methods. Another reason of finding no noun in these methods is that these methods use nouns that cannot be detected by the tagger, since the tagger has an accuracy of 97%.

4.5 Conclusion

The results suggest that nouns are present in the majority of method names. In 101014 of the 129583 methods a noun was found in the method name. This suggests that the noun is present in a lot of names and can therefore be important in the comprehension process of the method name.

If at least one noun occurs in both method name and in an identifier in the method implementation a co-occurrence of nouns is found in 83% of the methods. The results suggest that the co-occurrence of nouns could be useful, since a co-occurrence is present in a large amount of methods.

Our results suggest there is no specific noun based on the position in the method name that has a very strong relation with the implementation compared to the nouns on the other positions. This suggests that the assessment

of the co-occurrence based on single noun in the method name would not be sufficient.

A co-occurrence of nouns is found to be present often, however this results does not suggest methods without a co-occurrence of nouns are hard-to-comprehend. In the next chapter the comprehensibility of the methods is assessed. The results of the assessment might suggest if hard-to-comprehend method can be found using the co-occurrence of nouns.

Chapter 5

Comprehensibility of methods

5.1 Introduction

In the previous chapter the strength of the co-occurrence of nouns was research. Although there is a co-occurrence of nouns in 83% of the methods, we do not know if the presence of the co-occurrence of nouns is related to the comprehensibility of the method. Therefore the comprehensibility of the methods with and without the co-occurrence of nouns is inspected to verify the usefulness of the absence of the co-occurrence of nouns to find hard-to-comprehend methods.

The research question discussed in this chapter is:
What is the relation between the co-occurrence of nouns and the comprehensibility of the method?

5.2 Research method

Two samples are taken from the corpus to manually inspect the comprehensibility of the methods. The two samples both contain 100 random methods. One sample contains methods with a co-occurrence of nouns and the other sample contains methods without the co-occurrence of nouns. The methods from both samples have at least one noun in the method name and one noun in the implementation of the method.
We assess each method from the samples by trying to comprehend the

method by looking at the package name(s), class name(s), method name and implementation of the method. Both the method name and the identifiers in the implementation are assessed on comprehensibility. We simulate the maintenance task by only looking at the source code of the method itself during the comprehension process. While doing maintenance you do not want to access the source code of each other method that was invoked in the method, therefore we do not check the implementation of the invoked methods.
If we are not able to understand the method, we label the method as hard-to-comprehend. During the assessment of the comprehensibility of the method, we take two criteria into account. The method needs to be meaningful and appropriate, as discussed in section 1.6.

Although a method might be comprehensible, a co-occurrence of nouns might not be found. This can have various causes:

- The tagging algorithm is unable to detect the noun(s) and therefore no relation was found.
- The method name contains concepts inconsistent with the concepts used in the implementation.
- The method implementation contains meaningless identifiers
- The method name is too abstract to represent the implementation.

Therefore we check for these causes and count the times they occur.

In some cases it can be too hard for us to comprehend the source code. Therefore, we label methods with a cause if we cannot state a method is comprehensible for us with certainty. These causes are listed below.

- Can not understand the implementation. It was too hard for us to comprehend the implementation of the method due to a lack of knowledge. The method implementation was too large or we could not understand what the implementation was supposed to do.
- Depends on the domain/context. Concepts can be used specifically in a domain. Concepts can also mean something else in a different context. If the meaning of the concept was unclear, we labeled the method.

- Throwing only exception. Some methods only throw an exception. Since we cannot judge the comprehensibility of these methods, we labeled them accordingly.

By comparing the amount of hard-to-comprehend methods and the amount of occurred factors and properties with a negative impact on the comprehensibility found in both samples we discuss the relation between the co-occurrence of nouns and the comprehensibility of a method.

5.3 Results

The manual inspection of 100 methods without a relation and 100 methods with a relation given the results shown in table 5.1. The full list of inspected methods is shown in appendix B.

Criteria	Without co-occurrence	With co-occurrence
Can not understand the implementation	6	1
Comprehensible method	53	82
Hard-to-comprehend method	31	16
PartOfSpeech Error	6	1
Throws only exception	4	0
Total	100	100

Table 5.1: Comprehensibility of methods.

Criteria	Without co-occurrence	With co-occurrence
Inconsistent due to abbreviations	8	4
Depends on the domain	19	3
Name is very abstract	12	0
Meaningless identifiers	25	21
Missing verb	16	6
Inappropriate use of camelcase	5	0
Using vars starting with underscore	4	8

Table 5.2: List of counted factors and properties with a bad impact the comprehensibility of methods.

Since the results above have been collected based on our opinion, we show examples of the assessed methods to demonstrate the assessment. These examples include our argumentation used in the assessment.

5.3.1 Methods without co-occurrence of a noun

Examples of methods without a co-occurrence of nouns are discussed in this section. We discuss both comprehensible and hard-to-comprehend methods.

```
1  package org.jboss.ha.framework.server;
2
3  public class HAPartitionImpl{
4    public Vector getCurrentView()
5    {
6      Vector result = new Vector(this.members.size());
7      for (int i = 0; i < members.size(); i++)
8      {
9        result.add(((ClusterNode) members.elementAt(i)).getName());
10     }
11     return result;
12   }
13 }
```

We labeled this method as hard-to-comprehend, since the method name has absolutely nothing to do with the implementation. The implementation loops over all the 'members' and gets the name of the members. At the end the function returns a vector with the names of the members. When we expect to get a current view, we expect to receive something like a screen or an item on which something is displayed. We think a more appropriate name is 'getMemberNames'.

```
1  package org.jikesrvm.osr;
2
3  public class OSR_BytecodeTraverser {
4    private byte getReturnCodeFromSignature(String sig) {
5      byte[] val = sig.getBytes();
6
7      int i = 0;
8      while (val[i++] != ')') ;
9      return (val[i]);
10   }
11 }
```

Although this method above is quite comprehensible, it uses abbreviations('sig' and 'val') and therefore no relation can be detected. We argue 'sig' is a bad abbreviation, since it is not a common used abbreviation, else it would be used as abbreviation in the method name as well. Furthermore, a meaningless identifier 'val' is used. 'val' is probably used as the abbreviation for value. However, any variable contains a value, so 'val' or 'value' is argued to be meaningless.

```java
package org.argouml.uml.diagram.sequence.ui;

public class FigMessage{
  public Object getMessage() {
    return getOwner();
  }
}
```

We have labeled the method above as hard-to-comprehend, since we expected the method to return a message. This message is in this case of type object, which we think is too abstract to classify as a message. Furthermore, the name getOwner does not give us confidence, it will return a message either.

```java
package org.hibernate.collection;

public class PersistentMap {
  public void initializeFromCache(CollectionPersister persister,
      Serializable disassembled, Object owner) throws
      HibernateException {
    Serializable[] array = ( Serializable[] ) disassembled;
    int size = array.length;
    beforeInitialize( persister, size );
    for ( int i = 0; i < size; i+=2 ) {
      map.put(
          persister.getIndexType().assemble( array[i], getSession()
              , owner ),
          persister.getElementType().assemble( array[i+1],
              getSession(), owner )
        );
    }
  }
}
```

Cache is in this case no object in the software, but it is used as a term for memory. The PersistentMap object will be initialized with the data used from a CollectionPersister object. Since the name PersistentMap suggests the data is persistent and thus retrieved from memory from earlier use, we assess the name to be comprehensible.

5.3.2 Methods with co-occurrence of a noun

Examples of methods with a co-occurrence of nouns are discussed in this section. We discuss both comprehensible and hard-to-comprehend methods.

```java
package org.apache.xml.utils;

public class DOMBuilder {
  public void entityReference(String name) throws org.xml.sax.
      SAXException
  {
    append(m_doc.createEntityReference(name));
  }
}
```

The first example is a method that was labeled as hard-to-comprehend. Appropriate nouns are present, however we expect a verb to explain what is done with the nouns, since we still do not know what the method does by understanding the method name.

```java
package org.apache.tools.ant.taskdefs.condition;

public abstract class ConditionBase{
  protected final Enumeration getConditions() {
    return conditions.elements();
  }
}
```

The example above is a method how we expected every method to be. The method name explains the method implementation accurately.

```java
package fitnesse.http;

public class MockRequest{
  public void setRequestLine(String value) {
    requestLine = value;
  }
}
```

Although the method name is comprehensible and can be related to its implementation, more effort can be put in naming the identifiers. An identifier named 'value' is meaningless. Furthermore, renaming the identifier 'value' to newRequestLine would be appropriate. An appropriate parameter name would help if you use an IDE supporting auto completion. The auto completion tool shows the complete method declaration if part of the method is typed. Only if all the parameters of the method are named correctly, the programmers knows what value to pass to the method.

```
package fitnesse;

public class Shutdown{
  public RequestBuilder buildRequest() throws Exception {
    RequestBuilder request = new RequestBuilder("/?responder=
        shutdown");
    if (username != null)
      request.addCredentials(username, password);
    return request;
  }
}
```

If the implementation of the method above is right, it would be more appropriate to name the method 'buildRequestBuilder'. We argue that programmers can be confused by only 'Request' in the name and 'RequestBuilder' in the return type, since both concepts have a different meaning.

5.4 Discussion

In this section the results are discussed. Since we measured the comprehensibility of methods with and without relation, we discuss both methods.

We were unable to understand the method implementation of 7 of the 200 methods. This was due to the usage of mathematical calculations, bit-wise operations or large hard-to-comprehend program constructions. We ignored these methods in our analysis since it was impossible to properly assess the comprehensibility of the method names.

The methods called toString were found to be relevant observations. These methods return a string, but this string is not always stored in a temporary identifier. Furthermore, other identifiers used in the method were added to the string using the + operator, however no noun 'string' was used in the identifier information in these identifiers and therefore no co-occurrence of nouns could be detected. This shows that using only the identifier information from the method implementation in the analysis might be not sufficient.

Abbreviations in identifiers are widely used and did sometimes keep us from understanding the identifier without looking to more source code. An abbreviation such as 'blk' was found. This abbreviation can be interpreted in different ways, therefore abbreviations could confuse programmers during the maintenance task. The use of abbreviations was inconsistent between the method name and the words used in the identifiers in the implementa-

tion in a total of 12 methods. We found this to be a relevant finding, since Deißenböck and Pizka already suggested to use a single name for a single concept. This suggestion appears not only to be ignored in global, but also on method level.

Although, meaningless identifiers were found in 46 methods, the total method implementation was sometimes still comprehensible. The cause of this is that these identifiers did not always have an important role in the method implementation. An example of such an identifier is value. Comprehending such a name is easy. However, relating the name to its role in the implementation is hard, since the name is too general. You can call any variable "value", since a variable always represents a value.

In a total of 22 cases a verb was used inappropriate or it was totally missing in the method name. This influenced the comprehension process in most cases a lot, since we did not know what the method would do without inspecting the method implementation. This finding suggests that the work of Høst and Østvold is valuable since they checked for the use of a verb in a method name.

A remarkable note has to be made about following the Java Code Conventions. 12 of the 200 picked methods contained at least 1 variable starting with an underscore. The Java Code Conventions recommends to avoid this, so programmers do not always follow the Java Code Conventions.

Furthermore, in 5 of the 100 methods without a co-occurrence of nouns we found no noun was detected because camelCase was not used consistently in both method name and identifier names in the implementation. This caused the co-occurrence of nouns to be undetectable, since the identifiers could not be decomposed properly.

We found methods using or returning a constant value. These values were not always declared as constants and we did not know the meaning of the value. If these constant values were declared in constant variable, we might be able to understand the meaning of the value.

5.5 Conclusion

The results suggest that methods with a co-occurrence of nouns were comprehensible more often than methods without a co-occurrence of nouns. If no appropriate name is given to the method, it is harder to relate the method

name to the implementation. However, if no co-occurrence of nouns is detected, this does not strictly imply the method to be hard-to-comprehend. The results suggest that methods with a co-occurrence of nouns can be hard-to-comprehend as well.

The existence of a co-occurrence of nouns in a method is no guarantee the method is comprehensible either. However, the co-occurrence of nouns can be a good measurement to find methods that need further analysis. The manual verification is always necessary to ensure the method is comprehensible, since not all methods without co-occurrence of nouns are hard-to-comprehend.

The results promise that the co-occurrence of nouns may be used to predict hard-to-understand method names with a certain accuracy, but the actual accuracy is still an open question.

Chapter 6

Relating the co-occurrence of nouns to the size of a method

6.1 Introduction

In previous chapters we have analyzed the strength of the co-occurrence of nouns. We expect that the strength of the co-occurrence of nouns differs for methods based on their size. Our assumption is that the larger a method implementation, the more chance a co-occurrence of nouns is found, since more identifiers can be used in the implementation.
If a co-occurrence of nouns occurs more often in large methods, it might indicate that the co-occurrence of nouns is less useful for finding hard-to-comprehend methods. The results could also indicate we have to compensate for the size factor. If the strength of the co-occurrence is not related to the size or always the same, it might indicate the co-occurrence of nouns is useful in any method independent of its size.

The research question discussed in this chapter is:
What is the relation between the co-occurrence of nouns and the size of a method?

6.2 Research method

In section 3.1 the approach to measure the strength of the co-occurrence of nouns is discussed. The size of a method is measured by counting the SLOC(Source Lines Of Code) in the method implementation. The size of a method is measured by excluding lines of comment and removing lines containing only white spaces.
We only take a certain method size into account if at least 50 methods occur with this size. This makes the results more accurate, since we have a strength based on at least 50 methods.

6.3 Results

The results of our analysis are shown in graph 6.1.

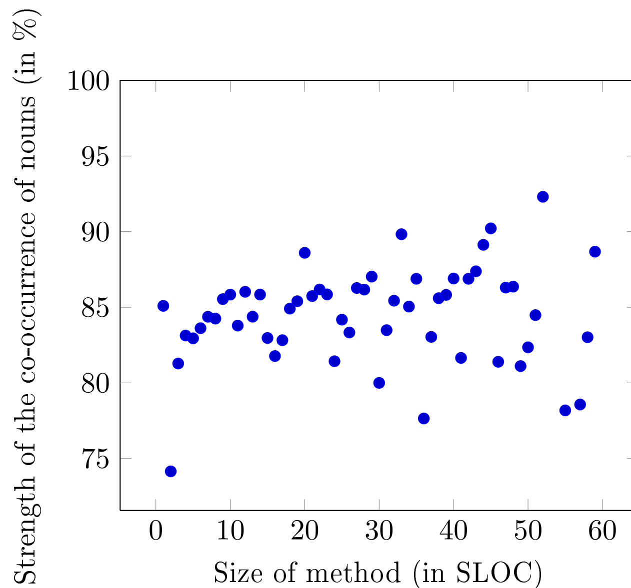

Figure 6.1: The relation between the strength of the co-occurrence of nouns and the size of the method implementation.

6.4 Discussion

The results suggest there is no strong negative or positive relation between the strength of the co-occurrence of nouns and the size of the method. The coordinates showed in graph 6.1 occur between a strength of 70% and 95%. Although we assumed the chance the co-occurrence of nouns is stronger if a method is larger, we found no evidence to support this claim.

We have furthermore only results of methods with a maximum unit size of 59. It would be interesting to research methods with a larger unit size. However, this requires a larger corpus since our corpus does not contain enough methods($>$50 methods) with a really large unit size(unit size of 60 or more).

6.5 Conclusion

We searched for a relation between the size of the method and the strength of the co-occurrence of nouns. However, no negative nor positive relation was found. Our hypothesis turned out to be false. The results suggest that hard-to-comprehend methods independent of their size can be found using the co-occurrence of nouns.

Chapter 7

Using the verbs to find the co-occurrences of nouns

7.1 Introduction

In chapter 4 a co-occurrence of nouns was found in over 83% of the methods with at least one noun in the method name and at least one noun in the identifiers in the implementation of the method. In this chapter we focus on more specific co-occurrences of nouns based on the verb in the method name.

Høst and Østvold found that programmers consistently use a specific verb in the method name if the method implementation contains certain statements and expressions. The combinations of these statements and expressions were called traceable attributes by Høst and Østvold. The set of attributes was extended by Singer et al.. Singer et al. called them fundamental nano-patterns.

We expect the verb in the method name to describe the action done using an object. We assume this object is represented by the noun in the method name(See section 1.2). We expect this verb-noun relation is present in the method implementation as well. The noun in the method name is expected to occur specifically in the identifier information in the nano-patterns in the method implementation. If this specific occurrence is present, it is useful to predict a possible noun to use in the method name.

The research question discussed in this chapter is:

Can the verb-noun relation in the method implementation be used to find the noun occurring in the method name?

7.2 Research method

The approach used is partially equal to the first step of our approach explained in chapter 3. The detection of the co-occurrence of nouns is based on results of earlier research by Høst and Østvold. Høst and Østvold generated a list of lexicon entries. A lexicon entry is a pair containing a verb and a set of nano-patterns occurring consistently in methods starting with this verb.

We base our research on the results of Høst and Østvold. To be sure our research produces valid results, we have to be sure to base our research on the right results. The research of Høst and Østvold differs on some aspects from ours:

- We analyze source code, while Høst and Østvold analyzed byte code.
- The corpus used by us is a subset of the corpus of Høst and Østvold.

Since we think these aspects can have an effect on our results, we replicate their research using source code analysis and a different corpus to generate a list of lexicon entries. Stoel supported the claim that the aspects listed above have an effect on the results found by relating the verb to the used nano-patterns[16].

Furthermore, we think the set of traceable attributes of Høst and Østvold is not sufficient to find all the identifier information needed to find a co-occurrence of nouns. Singer et al. extended the work of Høst and Østvold and identified a total of 17 nano-patterns. Because we value the set of nano-patterns of Singer et al. more, the set of nano-patterns of Singer et al. is used. The set of Singer et al. contains more nano-patterns, therefore it covers more behaviour of a method, for example reading and writing to local variables instead of detecting only reading local variables.

By measuring the strength of the co-occurrence of nouns for methods grouped by their verb and based on the identifier information found in the nano-patterns, we check if a noun from any of the identifier information found

in each nano-pattern is equal to the noun in the method name. Furthermore, do we manually assess one random method per verb from the set of methods without a co-occurring noun in the identifier information in the nano-patterns. This is only to demonstrate the usefulness of a co-occurrence based on the identifier information in the nano-patterns.

In the sections below we discuss each step of the approach in detail.

7.2.1 Finding commonly used verbs

The first step is finding the commonly used verbs in the methods. We search for them by analyzing all the methods from the corpus and count the times a verb occurs as the first word in the method name. This analysis is done using the decomposition and tagging algorithm described in section 3.1.3. We furthermore search for the commonly used verbs in methods containing at least one noun in the method name. Finding these methods is necessary since we want to measure the co-occurrence of nouns. Methods without a noun in the name cannot have a co-occurrence either.

7.2.2 Replicate lexicon entries for commonly used verbs

The second step is to replicate earlier research[7] of Høst and Østvold. The output of the replication is a generated list of lexicon entries[7] based on the verb in a method name and the consistent usage of nano-patterns in the method implementation. A list of nano-patterns occurring never(0%), seldom(<5%), rarely(<25%), often(>75%), very often(>95%) or always(100%) in methods with a specific verb is generated. These percentages were used by Høst and Østvold and are therefore used in this experiment.

7.2.3 Filtering inconsistently named methods

The list of generated lexicon entries is used to determine if a method contains an appropriate verb. Since we focus on comprehensible methods, we state that methods contain an appropriate verb if the method implementation contains all the nano-patterns shown in the list that occur often or more(>75%) in methods with this specific verb. The method should furthermore not contain nano-patterns from the list that occur rarely or less(<25%) in methods with this specific verb. We realize not all the methods with an

appropriate verb do conform to these specifications. However, using this consistency, there is a higher chance we use more properly named methods in our analysis[16]. An example to motivate this decision is shown in example 7.1.

```
1 package com.tonicsystems.jarjar;
2
3 public class JarJarTask extends AntJarProcessor
4 {
5     protected JarProcessor getJarProcessor() {
6         return new MainProcessor(patterns, verbose);
7     }
8 }
```

Figure 7.1: Example to motivate the need to filter the methods by the combination of verb and nano-patterns.

Although this method looks ok, it can be improve. It does not only 'get' a JarProcessor, but it also creates one. We therefore argue 'createJarProcessor' would be a more appropriate name for this method. The generated lexicon entries support our criticism. It states that methods starting with get rarely create objects. Since the method is inconsistently named, it is filtered.

7.2.4 Use lexicon entries to analyze identifier information

The final step of the research is to check the co-occurrence of nouns based on the identifier information used in the nano-patterns. In some nano-patterns identifiers can occur, while in other nano-patterns identifiers can not occur, since not only the use of certain statements and expressions can cause a nano-pattern to occur, but also the avoidance of certain statements and expressions. Depending on the nano-pattern, identifier information is extracted from the method implementation if the nano-pattern occurs often or more or rarely or less. One of the nano-patterns is NoReturn and is found if a method returns void. If a method returns void, it can not contain identifiers in a return statement. In this case, we check for identifiers if the pattern is not present. Another nano-pattern is LocalReader and is found if a method reads a local variable. If the nano-pattern LocalReader is present in a method, we check for the presence of local variables. Identifier information can only be found about the local variable if the nano-pattern is present.

7.3 Results

The first experiment is finding the most commonly used verb in method names. The results are shown in table 7.1. Only the 10 most commonly used verbs are listed.

Position	Verb	# Methods	% of total methods
1	get	34200	26.19
2	set	14408	11.03
3	is	5810	4.45
4	add	3858	2.95
5	create	3688	2.82
6	visit	1878	1.44
7	remove	1780	1.36
8	test	1462	1.12
9	has	1383	1.06
10	read	1040	0.80

Table 7.1: Most occurred verbs.

Since we research the co-occurrence of the nouns and therefore focus on methods with at least one noun, we have also researched method names starting with a verb and containing at least one noun. The 10 most commonly used verbs in these method names are shown in table 7.2.

Position	Verb	# Methods	% of total methods
1	get	32590	24.96
2	set	12614	9.66
3	is	4209	3.22
4	create	3114	2.38
5	add	2990	2.29
6	test	1364	1.04
7	visit	1354	1.04
8	has	1136	0.87
9	remove	1091	0.84
10	read	713	0.55

Table 7.2: Most occurred verbs in methods containing at least one noun.

Since the results in table 7.2 show that methods with the verbs: get, set, is, create or add are the most commonly used methods containing at least one noun, we analyzed these methods in more detail. We furthermore based the replication of the research of Høst and Østvold on this data. The results of this replication are shown in appendix C. In appendix D a comparison,

between the Lexicon Entries found by Høst and Østvold and the Lexicon Entries found by us, is shown.

In the sections below the results of measuring the strength of a co-occurrence of nouns for each group of methods are shown.

7.3.1 Methods starting with get

Methods starting with the verb get are the most commonly used methods found. The lexicon entries generated during the replication of earlier work suggest that methods starting with get seldom return void. Since this implies they often return something non-void, we check the identifier information in the return statement. The results of this experiment are shown in graph 7.2.

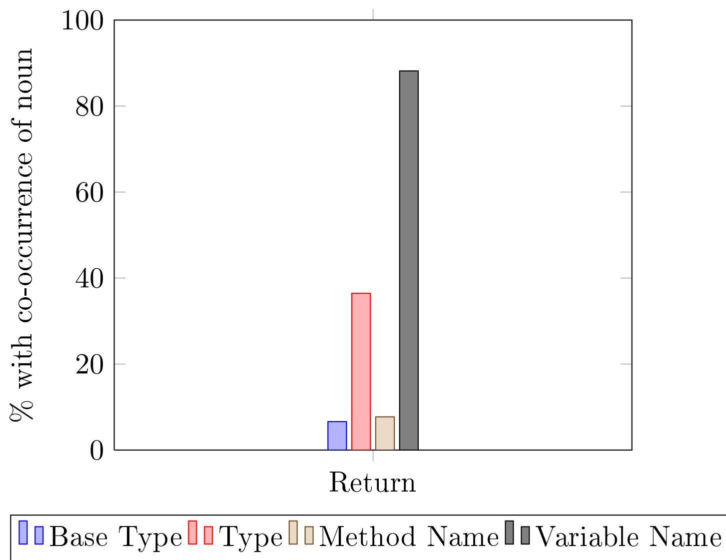

Figure 7.2: Co-occurrence of nouns in methods starting with get.

A total of 13618 methods starting with get conformed to the generated lexicon entries for a method starting with get. There was furthermore at least one identifier present in the return statement. No methods were found without a noun in the identifier information in the return statement.
In a total of 32 methods no co-occurrence of nouns was found between a noun in the method name and a noun found in the identifier information in

the return statement. An example of such a method is shown in example 7.3.

```
1 package com.tonicsystems.jarjar;
2
3 final class JikesRVMSocketImpl extends SocketImpl implements
      VM_SizeConstants {
4   public int getLocalPort() {
5     getLocalPortInternal();
6     return localport;
7   }
8 }
```

Figure 7.3: Example of a method starting with get without a co-occurrence of nouns based on the nano-patterns.

We argue this method is comprehensible, however it can be improved by using a camelCase style in the used variable. Improving this method in such a way would cause the tool to find a co-occurrence of nouns specifically in the return statement.

7.3.2 Methods starting with set

Methods starting with the verb set are the second most commonly methods found. The lexicon entries generated during the replication of earlier work suggest that methods starting with set often read the value of local variables and seldom have no parameters. Since this implies they often read the value of local variables and very often have parameters, we check the identifier information in the local variables that are read and the parameters of the method. The results of this experiment are shown in graph 7.4.

A total of 6469 methods starting with set conformed to the generated lexicon entries for a method starting with set. There was furthermore at least one identifier present as parameter and there was at least a local identifier that was used to read a value from.
There were 58 methods without a noun in the identifier information in the local reader pattern. There were also 58 methods without a noun in the identifier information from the parameters.
In a total of 910 methods no co-occurrence of nouns was found between a noun in the method name and a noun found in the identifier information in the nano-patterns. An example of such a method is shown in example 7.5.

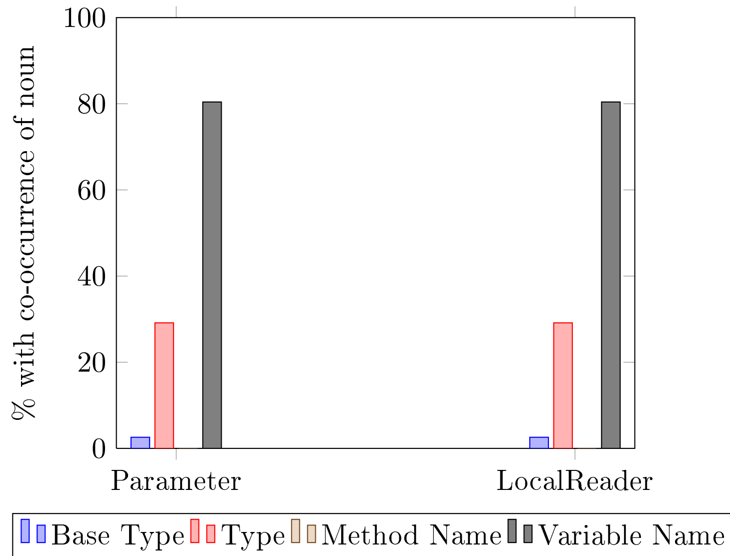

Figure 7.4: Co-occurrence of nouns in methods starting with set.

```
1 package org.jikesrvm.compilers.opt.ir;
2
3 public final class OPT_MethodOperand extends OPT_Operand {
4   public void setIsGuardedInlineOffBranch(boolean f) {
5     isGuardedInlineOffBranch = f;
6   }
7 }
```

Figure 7.5: Example of a method starting with set without a co-occurrence of nouns based on the nano-patterns.

We argue this method is comprehensible, however it can be improved by not using a meaningless variable as parameter. Improving this method could be done in the same way we suggested in 5.3.2. The parameter can be named 'newIsGuardedInlineOffBranch' to become comprehensible. This would also cause a co-occurrence of the nouns specifically in the local variable that is read and the parameter of the method to be present.

7.3.3 Methods starting with is

Methods starting with the verb is are the third most commonly used methods found. The lexicon entries generated during the replication of earlier work

suggest that methods starting with is seldom return void. Since this implies they often return something non-void, we check the identifier information in the return statement. The results of this experiment are shown in graph 7.6.

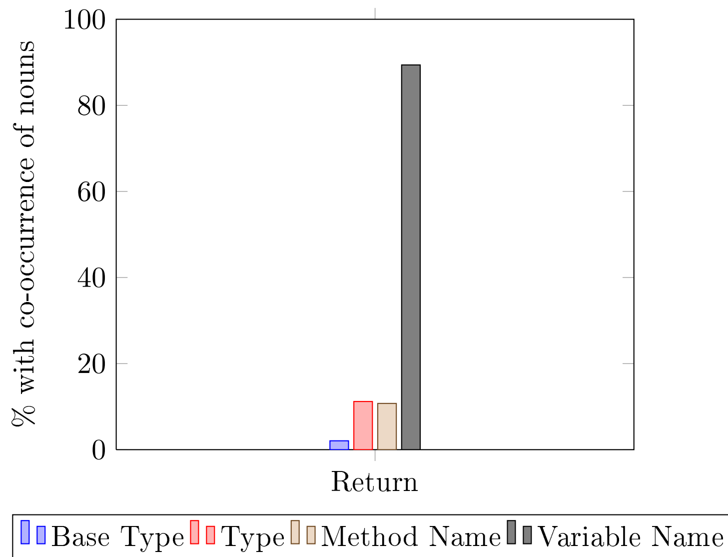

Figure 7.6: Co-occurrence of nouns in methods starting with is.

A total of 1555 methods starting with is conformed to the generated lexicon entries for a method starting with is. There was furthermore at least one identifier present the return statement.
There were no methods found without a noun in the identifier information in the return statement.
A single method was found without a co-occurrence of nouns between a noun in the method name and a noun found in the identifier information in the nano-patterns. This method is shown in example 7.7.

```java
package org.hibernate.loader.entity;

public class CascadeEntityJoinWalker extends
    AbstractEntityJoinWalker {
  protected boolean isJoinedFetchEnabled(AssociationType type,
      FetchMode config, CascadeStyle cascadeStyle) {
    return ( type.isEntityType() || type.isCollectionType() ) &&
        ( cascadeStyle==null || cascadeStyle.doCascade(
            cascadeAction) );
  }
}
```

Figure 7.7: Example of a method starting with is without a co-occurrence of nouns based on the nano-patterns.

We argue this method is comprehensible and it possibly does what the method name suggests. However, since we do not know what a JoinedFetch is, the assessment completely depends on our domain knowledge.

7.3.4 Methods starting with add

Methods starting with the verb add are the fourth most commonly used methods found. The lexicon entries generated during the replication of earlier work suggest that methods starting with add very often read the value of local variables, do rarely not invoke any methods and seldom have no parameters. Since this implies they often invoke methods and very often read the value of local variables and have parameters, we check the identifier information in the local variables that are read, invoked methods and the parameters of the method. The results of this experiment are shown in graph 7.8.

A total of 1216 methods starting with add conformed to the generated lexicon entries for a method starting with add. There was furthermore at least one identifier present as method, parameter and there was at least a local identifier that was used to read a value from.
There was one method without a noun in the identifier information in the local reader pattern. There were also 217 methods with methods that were invoked, but without a noun. 3 methods were found and had no noun in the identifier information from the parameters.
In a total of 115 methods no co-occurrence of nouns was found between a noun in the method name and a noun found in the identifier information in the nano-patterns. An example of such a method is shown in example 7.9.

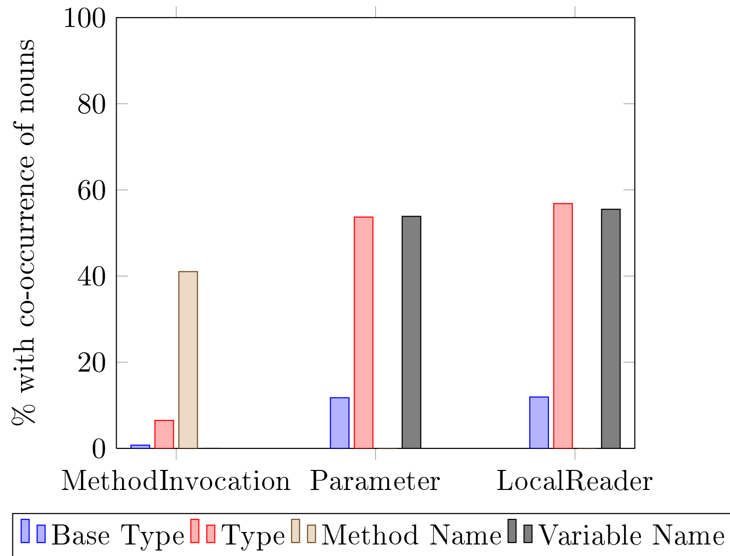

Figure 7.8: Co-occurrence of nouns in methods starting with add.

```
1  package org.jikesrvm.memorymanagers.mminterface;
2
3  public final class MM_Interface implements VM_HeapLayoutConstants,
      Constants {
4    /**
5     * Adds an object to the list of objects to have their
6     * <code>finalize</code> method called when they are reclaimed.
7     *
8     * @param object the object to be added to the finalizer's list
9     */
10   @Interruptible
11   public static void addFinalizer(Object object) {
12     Finalizer.addCandidate(ObjectReference.fromObject(object));
13   }
14 }
```

Figure 7.9: Example of a method starting with add without a co-occurrence of nouns based on the nano-patterns.

Without looking at the comment, we argue this method is hard-to-comprehend. We have no idea if 'object' is the finalizer that is added in the method, since the implementation suggests a candidate is added. The name of the parameter is 'object', which we think is a generic name. The implementation of the method does not suggest a Finalizer is added, but the documentation above the method does. We argue the method can be improved, since the implementation itself only suggests a candidate is added.

7.3.5 Methods starting with create

Methods starting with the verb create are the fifth most commonly used methods found. The lexicon entries generated during the replication of earlier work suggest that methods starting with create very often read the value of local variables, do rarely not invoke any methods and rarely return void. Since this implies they often invoke methods and return something non-void and very often read the value of local variables, we check the identifier information in the local variables that are read, invoked methods and the return statement of the method. The results of this experiment are shown in graph 7.10.

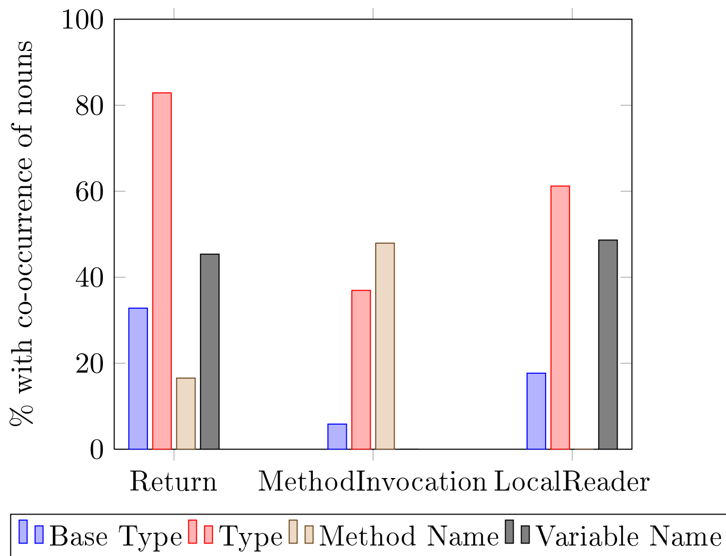

Figure 7.10: Co-occurrence of nouns in methods starting with create.

A total of 701 methods starting with create conformed to the generated lexicon entries for a method starting with create. There was furthermore at least one identifier present in the return statement, a local identifier that was used to read a value from and a invoked method in the method implementation. There were 58 methods without a noun in the identifier information in the local reader pattern. There were also 58 methods without a noun in the identifier information from the parameters. Furthermore, 217 methods with methods that were invoked, but without a noun were found.
In a total of 6 methods no co-occurrence of nouns was found between a noun

in the method name and a noun found in the identifier information in any of the nano-patterns. An example of such a method is shown in example 7.11

```
1  package org.tranql.ddl;
2
3  public class DDLBuilder {
4      public DDL createDrop(String entity) {
5          Table table = schema.getTable(entity);
6          if (null == table) {
7              throw new IllegalArgumentException("Entity " + entity +
                   " is undefined.");
8          }
9
10         DDL ddl = new DDL();
11         ddl.addChild(new Drop(table));
12         return ddl;
13     }
14 }
```

Figure 7.11: Example of a method starting with create without a co-occurrence of nouns based on the nano-patterns.

We argue this method name can be improved by naming the method 'createDLLWithDrop', since it creates a 'DDL' with a 'Drop' as child.

7.4 Discussion

The results suggest a lot of methods are filtered by only taking the methods with the consistently used nano-patterns into account. Filtering these methods enabled us to remove methods like the method in example 7.1.

However, the filters might have removed methods with patterns that do not occur that much anyway, independent of the verb. This might have caused methods were filtered that should have not been filtered. However, since no automatic assessment of names has been found yet, we assumed this was the best way to filter inconsistent named methods.

Furthermore, the method with a different verb, for example get or is, can both contain some equal nano-pattern(s). Methods starting with get or is both contain the nano-pattern NoReturn rarely or less. Therefore it can be hard to discriminate these methods from each other using only the NoReturn nano-pattern. All nano-patterns that did occur rarely or less and often or more were therefore useful to discriminate the methods starting with get and is from each other.

The results furthermore suggest that nouns causing the co-occurrence of nouns in methods can be found in specific nano-patterns and in specific identifier information. The methods starting with get, set, is or create are good examples to support this hypothesis. However, finding the occurred noun in methods starting with add was less useful. Since we have not manually analyzed a lot of these methods in detail, we do not know the cause of this.

7.5 Conclusion

We analyzed the methods containing one of the five most commonly used verbs. We searched for the co-occurrence of nouns in identifier information in the consistently used nano-patterns related to the verb in the method.

The results suggest that in methods starting with get, set, is or create any of the nouns in the method name co-occurred in specific identifier information in one of the nano-patterns in the method implementation. The noun in the method name co-occurred in one of the nano-patterns in the implementation in at least 80% of the methods.

The results suggests that this consistency was not found in methods starting with add. The co-occurrence of nouns in methods starting with add occurs in a less consistently amount of times. More research is needed to find the cause of this.

We have not verified if methods without the co-occurred noun in the identifier information in the consistently used nano-patterns are hard-to-comprehend or contain meaningless or inappropriate names in the large. However, we have found a relation between the combination of verb and noun in the method name and the nano-patterns and identifiers in the implementation. This relation might useful to suggest an appropriate noun to use in the method name. However, since no co-occurrence of nouns was found in all the results, more research needs to be done to verify this.

Chapter 8

Conclusion

We studied the relation between the co-occurrence of nouns and the comprehensibility of the method. We discussed a total of 4 sub research questions to get an answer on the following research question:
Can the co-occurrence of nouns be used to find hard-to-comprehend methods?

We measured the strength of the co-occurrence of nouns. The results suggest a co-occurrence of nouns is found in 83% of the methods. This suggests that a co-occurrence of nouns is present in a lot of methods and therefore the absence of the co-occurrence of nouns might be useful to indicate hard-to-comprehend methods.

Since the comprehensibility was unknown, we measured the comprehensibility of methods with a co-occurrence of nouns and without a co-occurrence of nouns. The results of this experiment suggests methods without a co-occurrence of nouns have a higher chance to be hard-to-comprehend. Furthermore, do the results suggest that the methods without a co-occurrence of nouns contain more properties and factors with a bad impact on the comprehensibility of methods than methods with a co-occurrence.

The size of a method implementation might be related to the amount of identifiers used in the methods. This might cause the size to have an effect on the strength of the co-occurrence of nouns. The results however suggest no negative nor positive relation is present and therefore the co-occurrence of nouns is useful find hard-to-comprehend method independent of the method size.

A more specific relation might exist between the nouns in the method name and implementation. We therefore searched for nouns in identifier information present in the consistently used nano-patterns based on the verb in the method name. In the methods with 4 of the 5 analyzed verbs a specific co-occurrence between a noun in the method name and a noun in the identifier information in a used nano-pattern was found. The results suggests a more precise co-occurrence of nouns can be found based on the verb in the method name.

Although more work needs to be done on this subject, we argue that the current results show that the co-occurrence of nouns can be used to find hard-to-comprehend methods. However, if a method without a co-occurrence of nouns is found, there is no guarantee this method is hard-to-comprehend. Not all the methods with a co-occurrence of nouns are comprehensible either. The manual verification of the comprehensibility is always needed. The co-occurrence of nouns is therefore only useful to find methods with a higher chance to be hard-to-comprehend.

Chapter 9

Future work

We focused primarily on methods containing a noun in the method name and a noun in the identifier information in the implementation. However, a noun does not occur in any method name. Future work can be done in assessing the mandatory existence of a noun in the method implementation if one is used in the method name.

In the last research question, we have focused on finding a co-occurrence of nouns based on the identifier information in the nano-patterns. However, no assessment has been done on the comprehensibility of methods if no appropriate noun was found in the identifier information.

Although we manually assessed the comprehensibility of the methods, this comprehensibility might be highly subjective. Therefore more research could be done on the global understanding of the assessed methods by all kinds of programmers.

It would also be interesting to see how useful an automatic assessment of the co-occurrence of nouns is for programmers. How many times do they really rename a method if it was proposed to be hard-to-comprehend? Or were the identifiers in the method implementation meaningless and are they adjusted by the programmer?

Bibliography

[1] Code conventions for the java programming language, April 1998. URL `http://www.oracle.com/technetwork/java/codeconv\-138413.html`.

[2] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001. ISSN 0018-9162.

[3] Emilio Collar and Ricardo Valerdi. Role of software readability on software development cost. 2008.

[4] Florian Deißenböck and Markus Pizka. Concise and consistent naming. In *In IWPC 2005*, pages 97–106. IEEE Computer Society, 2005.

[5] Xuefen Fang. Using a coding standard to improve program quality. In *Quality Software, 2001. Proceedings.Second Asia-Pacific Conference on*, pages 73 –78, 2001.

[6] Einar Høst and Bjarte Østvold. The java programmer's phrase book. In Dragan Gaševic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 322–341. Springer Berlin / Heidelberg. ISBN 978-3-642-00433-9.

[7] Einar W. Høst and Bjarte M. Østvold. The programmer's lexicon, volume i: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2880-5.

[8] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on ECOOP 2009 —*

Object-Oriented Programming, Genoa, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3.

[9] Edvard K. Karlsen, Einar W. Høst, and Bjarte M. Østvold. Finding and fixing java naming bugs with the lancelot eclipse plugin. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 35–38, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2.

[10] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168 –177, sept. 2009.

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

[12] Y. Matsuo and M. Ishizuka. Keyword extraction from a single document using word co-occurrence statistical information. *International Journal on Artificial Intelligence Tools*, 13:2004, 2004.

[13] J. Sametinger. A tool for the maintenance of c++ programs. In *Software Maintenance, 1990., Proceedings., Conference on*, pages 54 –59, nov 1990.

[14] Jeremy Singer, Gavin Brown, Mikel Luján, Adam Pocock, and Paraskevas Yiapanis. Fundamental nano-patterns to characterize and classify java methods. *Electron. Notes Theor. Comput. Sci.*, 253(7): 191–204, September 2010. ISSN 1571-0661.

[15] Michael M. Stark and Richard F. Riesenfeld. Wordnet: An electronic lexical database. In *Proceedings of 11th Eurographics Workshop on Rendering*. MIT Press, 1998.

[16] Jouke Stoel. Exploring the detection of method naming anomalies, 2012.

Appendix A

Example to demonstrate initial approach

```
1  package CH.ifa.draw.util;
2
3  public class VersionManagement {
4    public static String readVersionFromFile(String applicationName,
         String versionFileName) {
5      try {
6        FileInputStream fileInput = new FileInputStream(
             versionFileName);
7        Manifest manifest = new Manifest();
8        manifest.read(fileInput);
9
10       Map entries = manifest.getEntries();
11       // Now write out the pre-entry attributes
12       Iterator entryIterator = entries.entrySet().iterator();
13       while (entryIterator.hasNext()) {
14         Map.Entry currentEntry = (Map.Entry)entryIterator.next();
15         String packageName = currentEntry.getKey().toString();
16         packageName = normalizePackageName(packageName);
17         Attributes attributes = (Attributes)currentEntry.getValue()
               ;
18         String packageSpecVersion = attributes.getValue(Attributes.
             Name.SPECIFICATION_VERSION);
19         packageSpecVersion = extractVersionInfo(packageSpecVersion)
               ;
20         return packageSpecVersion;
21       }
22     }
23     catch (IOException exception) {
24       exception.printStackTrace();
25     }
26
27     // no version found
28     return null;
29   }
30 }
```

Figure A.1: Example to motivate not to count the times of occurrences.

This method name in example A.1 explains its implementation well, however none of the nouns used in the name is the most relevant noun in table A.1.

Noun	Counted co-occurrences of nouns
string	15
entry	14
name	9
version	9
iterator	8
package	8
file	7
manifest	7
attribute	6
input	5
spec	4
current	3
object	3
map	3
stream	3
exception	2
set	2
specification	2
void	2
value	2
application	1
info	1
io	1
key	1
stack	1
trace	1

Table A.1: Counted noun occurrences from the method implementation.

Appendix B

Analyzed methods on comprehensibility

B.1 Methods with co-occurrence of nouns

fitnesse.wikitext.widgets.TableRowWidgetTest.testPlainTextRow()
org.exolab.castor.xml.Marshaller.setSuppressXSIType(boolean)
org.apache.batik.css.engine.sac.CSSConditionalSelector.getSpecificity()
org.gjt.sp.jedit.syntax.ParserRule.createRegexpSequenceRule(char, int, java.lang.String, org.gjt.sp.jedit.syntax.ParserRuleSet, byte, boolean)
org.apache.xalan.lib.sql.DTMDocument.findGTE(int[], int, int, int)
fitnesse.http.MockRequest.setRequestLine(java.lang.String)
org.apache.xalan.extensions.ExtensionHandlerExsltFunction.getFunction(java.lang.String)
polyglot.ast.Import_c.name()
fitnesse.wikitext.Utils.replaceStrings(java.lang.String, java.lang.String[], java.lang.String[])
org.apache.commons.collections.ProxyMap.containsKey(java.lang.Object)
org.gjt.sp.jedit.bsh.NameSpace.unsetVariable(java.lang.String)
org.apache.tools.ant.taskdefs.optional.jlink.jlink.calcChecksum(java.io.File)
org.apache.xalan.lib.sql.DTMDocument.getNodeType(int)
org.argouml.model.AbstractCommonBehaviorHelperDecorator.setRecurrence(java.lang.Object, java.lang.Object)
org.apache.tools.ant.taskdefs.condition.ConditionBase.getConditions()
org.apache.xalan.xsltc.compiler.UseAttributeSets.addAttributeSets(java.lang.String)
org.jnp.interfaces.NamingContext.listBindings(java.lang.String)
org.exolab.castor.types.RecurringDurationBase.getDuration()

org.apache.tools.ant.types.selectors.BaseExtendSelector.getParameters()
org.apache.tools.ant.taskdefs.optional.jsp.JspC.createClasspath()
org.apache.tools.ant.types.DataType.isReference()
antlr.PythonCodeGenerator.genNextToken()
org.jikesrvm.compilers.opt.OPT_LocalCSE.getClassConstructor()
org.apache.batik.svggen.font.table.CmapFormat4.toString()
org.exolab.castor.persist.OID.getName()
org.jboss.resource.security.SecureIdentityLoginModule.login()
ognl.Ognl.isSimpleNavigationChain(java.lang.String)
org.apache.tools.ant.taskdefs.optional.perforce.P4Change.getEmptyChangeList()
org.jboss.proxy.generic.ProxyFactoryHA.getLoadBalancePolicy()
org.jboss.util.state.DefaultStateMachineModel.removeState(org.jboss.util.state.State)
fit.Parse.addToBody(java.lang.String)
org.argouml.uml.diagram.static_structure.layout.ClassdiagramNode.getFigure()
org.apache.tools.ant.taskdefs.optional.sos.SOSCheckin.buildCmdLine()
org.apache.tools.ant.Task.isInvalid()
polyglot.ast.AbstractNodeFactory_c.ClassDecl(polyglot.util.Position, poly-
glot.types.Flags, java.lang.String, polyglot.ast.TypeNode, java.util.List, poly-
glot.ast.ClassBody)
org.argouml.uml.diagram.UMLMutableGraphSupport.getSourcePort(java.lang.Object)
org.jikesrvm.classloader.VM_BytecodeStream.getFloatConstant(int)
org.gjt.sp.jedit.textarea.Gutter.isSelectionAreaEnabled()
org.jboss.resource.adapter.jms.JmsSession.createDurableSubscriber(javax.jms.Topic,
java.lang.String, java.lang.String, boolean)
org.hibernate.cfg.Configuration.getEntityNotFoundDelegate()
org.apache.batik.bridge.SVGGVTFont.createGlyphVector(java.awt.font.FontRenderContext,
java.text.CharacterIterator)
polyglot.ast.AbstractDelFactory_c.delCallImpl()
org.apache.xpath.axes.FilterExprIterator.filterExprOwner.setExpression(
org.apache.xpath.Expression)
org.jikesrvm.compilers.opt.ir.GuardedSet.setVal(org.jikesrvm.compilers.opt.ir.OPT_Instruction,
org.jikesrvm.compilers.opt.ir.OPT_Operand)
org.apache.xalan.xsltc.dom.NodeSortRecord.getNode()
org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean.afterPropertiesSet()
polyglot.ext.param.types.MuPClass_c.addFormal(polyglot.ext.param.types.Param)
antlr.MakeGrammar.addElementToCurrentAlt(antlr.AlternativeElement)
org.argouml.uml.diagram.static_structure.ui.FigClass.updateNameText()
fitnesse.responders.refactoring.RenamePageResponderTest.testDontRenameToExistingPage()
org.apache.tapestry.form.validator.ValidatorsBindingFactory.createBinding(
org.apache.tapestry.IComponent, java.lang.String, java.lang.String,

org.apache.hivemind.Location)
polyglot.ast.AbstractExtFactory_c.extCast()
org.argouml.uml.reveng.ui.RESequenceDiagramDialog.buildAction(java.lang.String)
org.argouml.ui.LookAndFeelMgr.getSmallFont()
org.exolab.castor.persist.ClassMolder.setIdentity(org.castor.persist.TransactionContext, java.lang.Object, org.exolab.castor.persist.spi.Identity)
org.hibernate.persister.entity.SingleTableEntityPersister.addDiscriminatorToSelect(org.hibernate.sql.SelectFragment, java.lang.String, java.lang.String)
org.apache.batik.util.PreferenceManager.getFiles(java.lang.String)
org.gjt.sp.jedit.gui.CompletionPopup.CandidateListModel.getSize()
polyglot.ast.AbstractDelFactory_c.delAmbQualifierNode()
org.jikesrvm.compilers.opt.ir.OPT_BC2IR.BasicBlockLE.isLocalKnown()
org.apache.commons.collections.map.MultiKeyMap.isEqualKey(org.apache.commons.collections.map.AbstractHashedMap.HashEntry, java.lang.Object, java.lang.Object)
java.lang.Class<T>.getEnumConstants()
fit.CannotLoadFixtureTest.testFixtureClassNotEndingInFixtureDoesNotExtendFixture()
CH.ifa.draw.contrib.Helper.getDrawingView(java.awt.Component)
fitnesse.Shutdown.buildRequest()
org.gjt.sp.jedit.MiscUtilities.objectsEqual(java.lang.Object, java.lang.Object)
org.jboss.varia.stats.CacheListener.ContentionStats.getMaxContentionTime()
org.exolab.castor.xml.schema.Wildcard.getProcessContent()
org.apache.bcel.generic.FieldGenOrMethodGen.removeAttribute(org.apache.bcel.classfile.Attribute)
org.gjt.sp.jedit.gui.DockableWindowManagerImpl.handleMessage(org.gjt.sp.jedit.EBMessage)
org.apache.commons.net.nntp.Article.getArticleNumber()
org.jikesrvm.compilers.opt.ir.MIR_Nullary.getClearResult(org.jikesrvm.compilers.opt.ir. OPT_Instruction)
org.springframework.ui.freemarker.FreeMarkerConfigurationFactory. setPreferFileSystemAccess(boolean)
org.apache.tools.ant.Diagnostics.printParserInfo(java.io.PrintStream, java.lang.String, java.lang.String, java.lang.String)
org.jboss.hibernate.jmx.Hibernate.setCacheProviderClass(java.lang.String)
org.hibernate.dialect.InformixDialect.getSelectSequenceNextValString(java.lang.String)
org.gjt.sp.jedit.textarea.BufferHandler.getReadyToBreakFold(int)
org.objectweb.asm.util.CheckMethodAdapter.checkDesc(java.lang.String, boolean)
polyglot.qq.QQ.parseDecl(java.lang.String, java.lang.Object, java.lang.Object, java.lang.Object, java.lang.Object, java.lang.Object, java.lang.Object, java.lang.Object, java.lang.Object)
org.apache.bcel.verifier.structurals.ExecutionVisitor.visitCASTORE(org.apache.bcel.generic.

CASTORE)
org.apache.tapestry.util.ComponentAddress.findComponent(org.apache.tapestry.IRequestCycle)
org.hibernate.hql.ast.tree.FromElementType.extractTableName()
org.apache.commons.httpclient.params.HttpConnectionParams.setSendBufferSize(int)
org.apache.xml.utils.DOMBuilder.entityReference(java.lang.String)
javax.management.StringValueExp.toString()
org.apache.commons.digester.Digester.error(org.xml.sax.SAXParseException)
polyglot.ast.ConstructorDecl_c.throwTypes(java.util.List)
org.argouml.model.mdr.CoreHelperMDRImpl.removeDeploymentLocation(java.lang.Object, java.lang.Object)
org.gjt.sp.jedit.buffer.JEditBuffer.getContextSensitiveProperty(int, java.lang.String)
org.jboss.remoting.loading.ClassByteClassLoader.addClass(org.jboss.remoting.loading.ClassBytes)
fitnesse.responders.WikiImportingResponderTest.testImportPropertiesGetAdded()
org.apache.batik.bridge.svg12.SVGMultiImageElementBridge.createGraphicsNode(org.apache.batik.bridge.BridgeContext, org.w3c.dom.Element)
org.jboss.web.tomcat.security.CustomPrincipalValve.UserPrinicpalRequest.setUserPrincipal(java.security.Principal)
org.exolab.castor.xml.util.XMLClassDescriptorResolverImpl.DescriptorCache.loadMapping(java.lang.String, java.lang.ClassLoader)
org.jboss.web.tomcat.security.CustomPrincipalValve.UserPrinicpalRequest.getMethod()
org.apache.xml.serializer.AttributesImplSerializer.getIndex(java.lang.String)
org.jboss.resource.adapter.jms.JmsMessage.setJMSCorrelationID(java.lang.String)
org.jboss.ejb.plugins.cmp.jdbc.bridge.JDBCAbstractCMPFieldBridge.getFieldType()
fitnesse.responders.editing.MergeResponder.makeInputTagWithAccessKey()
org.jboss.net.axis.transport.mailto.AbstractMailTransportService.closeStore(javax.mail.Store)

B.2 Methods without co-occurrence of nouns

org.jboss.mq.il.uil2.SocketManager.internalSendMessage(org.jboss.mq.il.uil2.msgs.BaseMsg, boolean)
org.springframework.orm.hibernate3.FilterDefinitionFactoryBean.getObjectType()
org.jikesrvm.classloader.VM_NormalMethod.genCode()
org.apache.velocity.runtime.parser.node.ASTAndNode.value(org.apache.velocity.context.InternalContextAdapter)
org.objectweb.asm.optimizer.MethodOptimizer.visitAnnotationDefault()
org.jikesrvm.compilers.baseline.ia32.VM_Compiler.genMonitorEnter()

org.mmtk.plan.Plan.harnessBegin()
org.jikesrvm.compilers.opt.ir.TrapIf.indexOfTCode(org.jikesrvm.compilers.
opt.ir.OPT_Instruction)
org.jruby.runtime.builtin.meta.AbstractMetaClass.Meta.getUndefineMethods()
org.apache.xalan.lib.ExsltSets.intersection(org.w3c.dom.NodeList, org.w3c.dom.NodeList)
org.apache.xalan.xsltc.compiler.StepPattern.analyzeCases()
org.jikesrvm.osr.OSR_BytecodeTraverser.getReturnCodeFromSignature(java.lang.String)
org.apache.tapestry.valid.DateValidator.toObject(org.apache.tapestry.form.
IFormComponent, java.lang.String)
org.exolab.castor.gui.QueryAnalyser.MainFrame.openDB()
polyglot.ast.ArrayAccess_c.prettyPrint(polyglot.util.CodeWriter, polyglot.visit.PrettyPrinter)
polyglot.ast.Assert_c.typeCheck(polyglot.visit.TypeChecker)
org.springframework.aop.framework.ReflectiveMethodInvocation.getStaticPart()
org.jikesrvm.VM.sysWriteln(java.lang.String, int, java.lang.String, int)
org.argouml.language.java.generator.JavaRecognizer.classTypeSpec()
org.argouml.uml.ui.behavior.common_behavior.ActionNewCreateAction.getInstance()
org.argouml.model.UUIDManager.getNewUUID()
org.apache.xalan.lib.sql.DTMDocument.getNamespaceURI(int)
org.argouml.uml.diagram.use_case.UseCaseDiagramGraphModel.getOwner(java.lang.Object)
fitnesse.slim.SlimClient.sendBye()
org.apache.tapestry.parse.ParseMessages.rangeError(org.apache.tapestry.parse.TemplateToken,
int)
org.jboss.ha.framework.server.HAPartitionImpl.getCurrentView()
org.springframework.core.io.UrlResource.getFilename()
org.argouml.uml.ui.TabTaggedValuesModel.addRow(java.lang.Object[])
org.argouml.persistence.ArgoParser.handleAuthorEmail(org.argouml.persistence.XMLElement)
org.jikesrvm.compilers.opt.ir.Nullary.indexOfResult(org.jikesrvm.compilers.opt.ir.
OPT_Instruction)
org.jikesrvm.compilers.opt.ir.IfCmp2.hasTarget1(org.jikesrvm.compilers.opt.ir.
OPT_Instruction)
org.springframework.util.comparator.BooleanComparator.toString()
org.apache.batik.ext.awt.image.rendered.GaussianBlurRed8Bit.surroundPixels(double,
java.awt.RenderingHints)
org.hibernate.collection.PersistentMap.initializeFromCache
(org.hibernate.persister.collection.CollectionPersister, java.io.Serializable, java.lang.Object)
fitnesse.fixtures.ResponseExaminerTest.setup()
org.hibernate.sql.QueryJoinFragment.addCrossJoin(java.lang.String, java.lang.String)
org.jboss.jmx.connector.rmi.RMIConnectorImpl.getClassLoaderFor(
javax.management.ObjectName)
antlr.LLkParser.traceOut(java.lang.String)

org.apache.tools.ant.types.XMLCatalog.InternalResolver.resolveEntity(java.lang.String, java.lang.String)
org.jikesrvm.compilers.opt.OPT_BURS_Common_Helpers.PLLRL(
org.jikesrvm.ArchitectureSpecific.OPT_BURS_TreeNode)
org.apache.batik.swing.gvt.JGVTComponent.releaseRenderingReferences()
org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler.
includeReadAttribute(java.lang.reflect.Method, java.lang.String)
org.springframework.jdbc.object.SqlFunction.runGeneric()
org.apache.xalan.templates.Stylesheet.getXSLToken()
fitnesse.components.Base64Test.testDecodeNothing()
org.apache.bcel.verifier.structurals.Frame.getClone()
org.jboss.logging.appender.FileAppender.Helper.makePath(java.lang.String)
org.apache.batik.ext.awt.image.codec.PNGImageEncoder.writeIEND()
fitnesse.updates.FileUpdate.shouldBeApplied()
org.gjt.sp.jedit.gui.VariableGridLayout.getLayoutAlignmentX(java.awt.Container)
org.jikesrvm.memorymanagers.mminterface.SynchronizationBarrier.resetRendezvous()
org.hibernate.collection.PersistentElementHolder.needsUpdating(
java.lang.Object, int, org.hibernate.type.Type)
org.gjt.sp.jedit.gui.statusbar.MemoryStatusWidgetFactory.MemoryStatus.getToolTipText()
org.argouml.uml.diagram.ui.ModeCreateGraphEdge.isConnectionValid(
org.tigris.gef.presentation.Fig, org.tigris.gef.presentation.Fig)
org.jikesrvm.compilers.opt.ir.MIR_CondMove.getValue(org.jikesrvm.compilers.opt.ir.
OPT_Instruction)
org.hibernate.criterion.SimpleProjection.getColumnAliases(int)
org.apache.batik.css.parser.CSSLexicalUnit.getIntegerValue()
org.jikesrvm.compilers.opt.OPT_SpaceEffGraphNode.InEdgeEnumeration.hasMoreElements()
org.tranql.field.IdentityExtractorAccessor.getFieldClass()
fitnesse.responders.run.ExecutionLogTest.testPageLink()
org.apache.tapestry.web.WebContextResource.getLocalization(java.util.Locale)
org.apache.tools.ant.taskdefs.optional.ccm.Continuus.getCcmCommand()
org.jikesrvm.compilers.opt.ir.Attempt.indexOfOffset(org.jikesrvm.compilers.opt.ir.
OPT_Instruction)
org.argouml.ui.explorer.rules.GoStateMachineToState.getRuleName()
org.jruby.RubyObject.isKindOf(org.jruby.RubyModule)
org.apache.batik.svggen.XmlWriter.writeChildrenXml(org.w3c.dom.Attr,
org.apache.batik.svggen.XmlWriter.IndentWriter)
org.argouml.uml.diagram.ui.FigSingleLineText.removeFromDiagram()
org.exolab.castor.types.GDay.getValues()
org.jikesrvm.compilers.opt.OPT_SpaceEffGraphNodeList.nextElement()
org.objectweb.asm.tree.LookupSwitchInsnNode.getType()

com.microstar.xml.XmlParser.tryRead(java.lang.String)
org.apache.commons.lang.mutable.MutableInt.hashCode()
org.apache.xpath.NodeSetDTM.getLength()
org.apache.batik.util.RunnableQueue.getIteratorLock()
polyglot.ast.CharLit_c.typeCheck(polyglot.visit.TypeChecker)
org.apache.commons.codec.language.Metaphone.regionMatch(java.lang.StringBuffer, int, java.lang.String)
org.gjt.sp.util.StandardUtilities.getLeadingWhiteSpaceWidth(java.lang.CharSequence, int)
org.apache.batik.ext.awt.MultipleGradientPaintContext.getAntiAlias(float, boolean, float, boolean, float, float)
org.gjt.sp.jedit.pluginmgr.MirrorListHandler.peekElement()
org.jikesrvm.compilers.common.assembler.ia32.VM_Assembler.emitPatchPoint()
org.jikesrvm.compilers.opt.ir.Prepare.indexOfOffset(org.jikesrvm.compilers.opt.ir.OPT_Instruction)
org.apache.bcel.generic.ObjectType.referencesInterfaceExact()
org.jboss.console.plugins.monitor.ManageStringThresholdMonitorServlet.doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)
org.apache.commons.httpclient.params.HttpConnectionParams.getReceiveBufferSize()
javax.management.openmbean.CompositeType.isValue(java.lang.Object)
fit.FitServerTest.testFitParseExceptionDontCrashSuite()
org.argouml.uml.diagram.sequence.ui.FigMessage.getMessage()
org.springframework.orm.hibernate3.AbstractSessionFactoryBean.getObject()
org.jikesrvm.compilers.opt.ir.Multianewarray.getDimension(org.jikesrvm.compilers.opt.ir.OPT_Instruction, int)
org.apache.commons.collections.map.LRUMap.moveToMRU(org.apache.commons.collections.map.AbstractLinkedMap.LinkEntry)
fitnesse.responders.editing.SaveResponderTest.testKnowsWhenToMerge()
org.argouml.uml.ui.foundation.core.EnumerationListModel.targetSet(org.argouml.ui.targetmanager.TargetEvent)
org.springframework.web.servlet.view.tiles.TilesConfigurer.afterPropertiesSet()
org.gjt.sp.jedit.syntax.XModeHandler.findParent(java.lang.String)
org.gjt.sp.jedit.textarea.FirstLine.toString()
org.apache.xalan.lib.sql.DTMDocument.getDeclHandler()
org.jboss.ejb.plugins.cmp.jdbc2.bridge.JDBCCMRFieldBridge2.isSingleValued()
fitnesse.responders.run.TestExecutionReportTest.tablesShouldBeDeserialized()
org.apache.commons.lang.text.StrSubstitutor.checkCyclicSubstitution(java.lang.String, java.util.List)

Appendix C

Found lexicon entries

Below a list of the lexicon entries found by replicating the work of Høst and Østvold is shown.

C.1 Methods starting with get

They rarely use one or more control flow loops, writes values to local variables, creates new objects, uses type casts or instanceof operations, throws exceptions, calls another method with the same name. They seldom writes values to (static or instance) field of an object, reads values from an array, writes values to an array, creates a new array, calls itself recursively, return void.

C.2 Methods starting with set

They very often read values of local variables, return void. They often have no branches in method body. They rarely write values to local variables, create new objects, read (static or instance) field values from an object, use type casts or instanceof operations, throw exceptions, call another method with the same name. They seldom use one or more control flow loops, have no parameters, read values from an array, write values to an array, create a new array, call itself recursively.

C.3 Methods starting with is

They often have no branches in method body. They rarely use one or more control flow loops, write values to local variables, use type casts or instanceof operations, call another method with the same name. They seldom create new objects, write values to (static or instance) field of an object, read values from an array, write values to an array, throw exceptions, create a new array, call itself recursively, return void.

C.4 Methods starting with create

They often read values of local variables. They rarely do not issue any method calls, use one or more control flow loops, write values to (static or instance) field of an object, write values to an array, use type casts or instanceof operations, create a new array, return void, call another method with the same name. They seldom read values from an array, call itself recursively.

C.5 Methods starting with add

They very often read values of local variables. They often return void. They rarely do not issue any method calls, use one or more control flow loops, write values to (static or instance) field of an object, read values from an array, write values to an array, use type casts or instanceof operations, throw exceptions. They seldom have no parameters, create a new array, call itself recursively.

Appendix D

Comparison of lexicon entries

D.1 Methods starting with get

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. Specializations of get are is and size. A somewhat related name is hash.

Although Høst and Østvold found that get methods often read state we have found otherwise. But we found that get methods do return void rarely, like Høst and Østvold and we therefore only focus on the return statement.

D.2 Methods starting with set

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. Generalizations of set are handle and initialize. Somewhat related names are accept, visit, end and insert.

Since Høst and Østvold probably did not measure parameters as the use of local variable, they have measured that set methods rarely use local variable. However, we do not agree with this, since we interpret parameters as local

variable and therefore found that local variable are read often. We have not found that set methods do manipulate state often.

D.3 Methods starting with is

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. Generalizations of is are has and get. Somewhat related names are accept, visit, hash and size.

We have found that is methods seldom return void. Høst and Østvold found is methods rarely return void, but this suggests that we have to measure the return statement of the is method.

D.4 Methods starting with create

create. Among the most common method names. Methods named create very often create objects. Furthermore, they rarely call methods of the same name,

A quite remarkable result was found when we analyzed the create methods. Høst and Østvold found that create methods very often create objects. However, we found that create methods do not create object in more than 66% of the times. However, we found that create methods do often call methods and return a value other than void and write to local variables very often. We think this difference can be explained because Høst and Østvold analyzed Java bytecode and we analyze Java source code. When source code is transformed to bytecode certain optimization are done, such as method inlining. Using method inlining the implementation of a method is transformed such that invoked method calls are replaced by the implementation of the method itself. Therefore our analysis might detect method calls and the analysis of Høst and Østvold detected the creation of an object.

D.5 Methods starting with add

add. Among the most common method names. Methods named add often read state. Similar names are remove and action.

We found that add methods very often have parameters and read local variable. They furthermore often call other methods. Høst and Østvold did not find add methods do have parameters very often. Since we assume they did not detect the reading of values of parameters, they probably did not detect local variable were read. However, they found add methods do read state often, but we did not.