Exploring the Detection of Method Naming Anomalies

Jouke Stoel

(23-07-2012)

Master Software Engineering

Supervisor : dr. Jurgen Vinju

Universiteit van Amsterdam

Contents

Abstract

Program comprehension is a major part of software maintenance. Earlier research has shown that identifiers have a big impact on comprehensibility. For instance, using the wrong name for a method leads to spoiled comprehension in numerous other places. Given this, it is strange that there is little support for analysing the used names.

In this research we will investigate the automatic detection of method implementations that are in conflict with their given names. The goal is to get a deeper insight into these so called Method Naming Anomalies.

In order to analyse the method implementation we turn to previous research of Høst and Østvold who used nano patterns, simple properties exhibited by the code, to compare methods with each other. We extract these nano patterns from the source code, earlier research used byte code for this extraction. Methods that contain different nano patterns compared to other methods with similar names, are considered outliers.

By evaluating the found outliers we find that the method that was presented by Høst and Østvold does quicker lead to identifying methods that have bad names compared to when we take a random selection of methods.

Next to this we propose an alternate approach in finding these method naming anomalies using Formal Concept Analysis. By making use of Formal Concept Analysis we can locate methods which contain rare combinations of nano patterns. We show that the methods that get identified by this method also have a higher change of being methods that have a bad name. While both outlier location methods have overlap regarding the found outliers, there are also differences in found outliers.

Preface

The last three years of studying was an intense but very satisfying ride. I am very grateful that I was able to experience it. I found that studying besides a job is not always easy. This especially counts for the final phase but this thesis is a testimony that even difficult tasks come to a close.

Special thanks goes out to my tutor Jurgen Vinju for his guidance during the last period. His deep knowledge and endless optimism was of great support.

Next I would like to thank the other tutors of the Software Engineering Master of the Universiteit van Amsterdam, Paul Klint, Hans Dekkers, Tijs van der Storm, Jan van Eijck and Hans van Vliet. I have learned a lot during their classes.

Another big thanks goes out to all the other persons in the SEN-1 research group of CWI. Conversations at the coffee machine or over lunch helped in getting new cool ideas or discard rubbish once. Being able to work in such an environment is very inspiring.

I would also like to thank Jeremy Singer of the University of Glasgow for giving us the source code of the nano pattern analyser they used in their research into nano patterns. This enabled us to do a deeper analysis on some of our found results.

I my three years of studying I have met a lot of nice fellow students. Some however I feel I owe special thanks to.

First of all Christian Koeppe. We have spend a lot of time working together on various assignments and he was a big help during the writing of this thesis. Next I would also like to extend my gratitude to Arie van der Veek and Davy Meers for, like Christian, we spend a lot of time together finishing various courses. I feel that we were a good team.

I would also like to mention my fellow students Denis van Leeuwen en Luuk Stevens. Both also dived into the world of naming. The discussions we had and the exchange of ideas was very fruitful.

I also owe a lot of gratitude to my employer, Ordina for making it possible for me to study. I think that there are not many companies that are so lenient and relaxed about employees that are also part-time students.

And last but most important I am very grateful to my girlfriend Sara who has endured the last three years with flying colors. She has been of invaluable support —thank you.

Chapter 1

Introduction

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

Martin Fowler

It is widely recognized in the field of software engineering that it is important to write code that is easily understood. Already in 1971 this was recognized by Weinberg who wrote about 'ego less programming' [28]. In his book Weinberg states that the central objective of ego less programming is "making the program clear and understandable to the person or people who would ultimately have to read it." In more recent years this has been underlined by experts in the field like McConnell who emphasised that developers should "write programs for people first, computers second" [21].

Even though the influence of readability is widely recognized this does not mean that it is an easy task. This is illustrated by the amount of research that has been done in this field. For instance, Rugaber note that reading code is the most time consuming part of program understanding [24]. Buse and Weimer propose the use of a specialised readability metric to measure the readability of a program [3].

1.1 The importance of identifiers

Deissenboeck and Pizka investigated the ECLIPSE[1] source code and showed that identifiers make up for more than 70% of all the characters [8]. Other investigations of other applications show similar results. Although code consists of more then identifiers —like keywords, delimiters, operators and literals— claiming that identifiers are an important

[1]Eclipse is a open-source JAVA IDE which is widely used by JAVA developers (http://www.eclipse.org, version 3.3.1)

source of information is not far fetched. These claims are backed by research of Lawrie et al. who showed the impact of identifier names on program comprehension [18].

1.2 Coming up with the right identifier name

If identifiers are that important it is strange that there is little support for choosing the right name while developing. The only guidance that is given is on the syntactical element of naming or contain vague descriptions on the meaning of a name. For instance, on the convention of method names the original Sun Java Code Conventions state that: "Method should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized" [22].

While convention of syntax is important it does not help a developer in choosing a name which is meaningful. Many times the advice is given that a name should be meaningful, guidance on *how* to determine meaningfulness is not supplied. It is up to the developer to determine whether or not a name is suitable for the given situation.

1.3 How to define *meaning*?

In an attempt to tackle this problem Høst and Østvold turn to Wittgenstein[2] [14]. According to Wittgenstein the meaning of a word is derived from the way it is used [29]. The word itself does not have an "abstract" or "objective" meaning or, as remarked by Høst and Østvold, the meaning of a word is determined by the sum of its use [14].

Using this notion the meaning of an identifier could be approximated by analysing its use. By doing this we should be able to find the common properties of an identifier which are embedded in its usage. We have to accept that this approximation will always be imperfect since we are incapable to analyse all different uses of the identifier and new uses will always be added [14].

1.4 Consistent naming

Deissenboeck and Pizka note that meaningfulness alone is not enough, a name should be used *consistent* [8]. It should consistently refer to the same concept the programmer wants to express. Inconsistent naming use increases the effort of a programmer to understand a program [8].

The possibilities of automatically checking for consistent identifier usage by analysing the used identifier names has been subject of earlier research [8, 17, 27]. In most of this research the focus lay on comparing identifier names with each other[3]. The usage, or *how* the identifiers were used, was not taken into account

[2]Wittgenstein was a language philosopher who lived between 1889 and 1951 (http://en.wikipedia.org/wiki/Ludwig_Wittgenstein)

[3]In the research of Thies and Roth type information of the identifier was also taken into account [27]

Høst and Østvold present a novel way of checking consistent method naming by relating the method name to its implementation [14]. They use the notion of approximated meaning as described earlier. By identifying the presence of certain *nano patterns* in the methods byte code they are able to check if an implementation was consistently used with methods that share similar names. Inconsistencies between the name and the implementation is called a *Method Naming Anomaly*.[4]

The difference between the approach of Høst and Østvold and others is that they used the information embedded in the method body itself to find inconsistent method names.

1.5 Research questions

Earlier research of Høst and Østvold showed that it is possible to check whether method names were used consistently by comparing their implementations [14]. Methods that had inconsistent implementations were marked as anomalies.

A question remaining is what these found anomalies tell us about the code. Without a deeper understanding it is hard to reason about their severity. By reproducing the research of Høst and Østvold we want to get more insight in the nature of these method naming anomalies. Therefore the central goal of our research will be acquiring a deeper understanding of Method Naming Anomalies and how they can be located

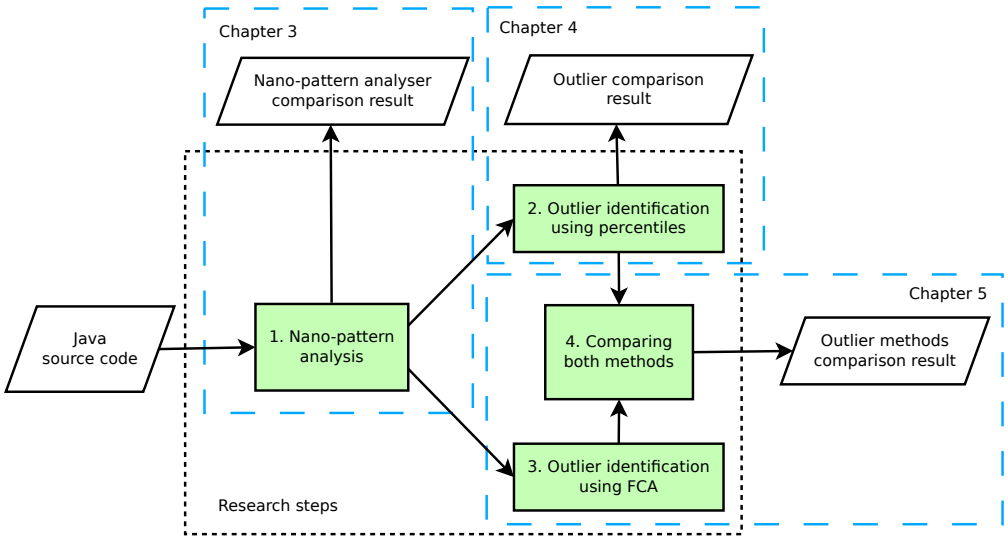


Figure 1.1: Research framework and organisation

To get this deeper insight we have split up our research into three different parts (see Figure 1.1). In the first part we focus on the nature of nano patterns and how they can be located in the source code (chapter 3). We do this by building a source code nano

[4]Høst and Østvold use the term Method Name Bug

pattern analyser and comparing it to the byte code nano pattern analyser of Singer et al. [26]. In this part we want to answer the questions:

Question 1. *How does our source code nano pattern analyser compare to the byte code analyser created by Singer et al.?*

In the second part we take a deeper dive into identifying method naming anomalies using the method of Høst and Østvold [14] (chapter 4). By doing this we want to get a better insight on the nature of method naming anomalies and the type of methods that get identified when using their method. In this part we want to answer the following question:

Question 2. *Do the methods that get identified using the method of Høst and Østvold have bad names?*

In the third part we present an alternate approach in finding method naming anomalies (chapter 4). By making use of *Formal Concept Analysis* we want to access the rare combinations of nano patterns that methods exhibit as a possible method of finding naming anomalies. The question that we want to answer in this part is:

Question 3. *How does our FCA method of identifying method naming anomalies compare to the method of Høst and Østvold?*

To summarize, this thesis makes three main contributions:

- It presents a nano pattern analyser based on the actual source code instead of the byte code (chapter 3).
- It describes the influence of different boundary values while locating method naming anomalies using the original method as introduced by Høst and Østvold (chapter 4).
- It presents a different approach for locating method naming anomalies using Formal Concept Analysis (chapter 5).
- It compares the results found between the original locating method and our presented method (chapter 5).

1.6 Scope

In our research we will focus on method names and their implementations in Java code. We have chosen to analyse Java code because it is widely available and is used in earlier research which we want to reproduce [14].

We will focus on the first part of the method name. This is often a verb, but not always. According to the Java Code Conventions it should however contain the action-oriented part of the name [22]. That is why we refer to this first part of a method name as the *action-token*.

The reason why we chose to analyse the first token of the method name and not the complete name is twofold. The first reason is simplicity. Comparing method names based on the same first token is easy. Comparing method names based on the complete name is harder. In the first case it is suffice to do a check on equality of the token, in the second case it would mean that we would have to decompose the whole name and analyse the semantics of the name using a mechanism like part-of-speech tagging.

Next to this Høst and Østvold have shown that by grouping methods based on the first token of their name and analysing their implementations we can analyse the behaviour that is shared amongst methods starting with the same token [13].

1.7 Organisation

Chapter 2 contains the background of our research and summarizes related work. Chapter 3 describes the source code analyser and the comparison with the byte code analyser of Singer et al. [26]. In chapter 4 we investigate and reproduce earlier work by Høst and Østvold described in their paper 'Debugging Method Names' [14]. In chapter 5 we present our own novel approach on finding method name anomalies and we compare it with the results found in chapter 4. Chapter 6 contains a discussion and an overview of possible future work.

Chapter 2

Background

Carefully breaking a computation into methods and carefully choosing their names communicates more about your intentions to a reader then any other programming decision.

Kent Beck

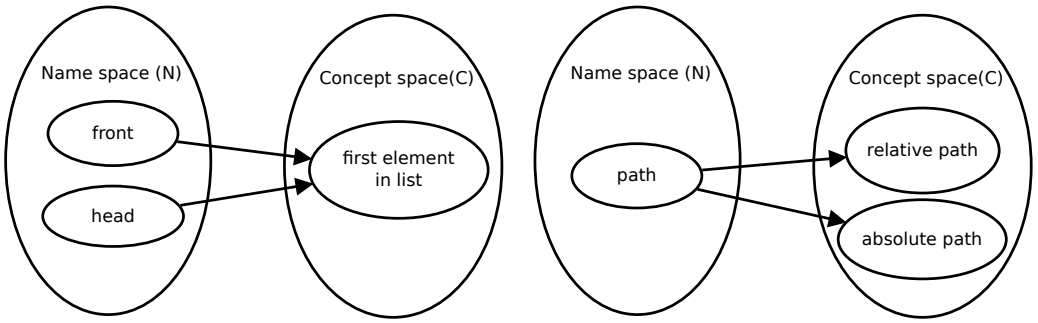
2.1 The 'other' language

Next to the computer language that is used in an application another language exists. This language is used for naming the identifiers [19]. The language describes the application domain and contains a vocabulary to identify the different concepts that are present in the domain.

Opposed to a computer language this language is informal and is not imposed by the compiler. Therefore it holds no meaning for the compiler. The sole purpose of this language is to make the application understandable for people. For example, a compiler does not differentiate between an identifier that is called 'Xasdf13kjt' and one that is called 'findPerson'. For us humans this is a big difference. Without proper naming understanding an application is much harder. Evidence for this claim is the existence of so called 'Code Obfuscaters'; applications that deliberately change the names of identifiers to meaningless labels so that the code is harder to understand and its content is protected [20].

2.2 Why consistent identifier naming is important

This 'other' language consists of a vocabulary with which we can identify the different concepts that are applicable in our application domain. For instance, in a banking



(a) Example of a synonym (b) Example of a homonym

Figure 2.1: Examples of synonyms and homonyms as described by Deissenboeck and Pizka, Lawrie et al. [8, 17].

application this can be concepts like *Account* or *Currency* but also can consist of technical concepts like *List* or *Controller* or even operations like *addAccountToList*, *getCurrency* or *findPersonWithAccount*.

Using the same name for the same concepts throughout an application is important for the comprehensibility [8]. By using the same names for the same concepts the learning effort needed by a new maintainer or developer is reduced. Different names for the same concepts, *synonyms*, needlessly enlarge the total vocabulary of the application language and increase the needed learning effort [8] (see 2.1(a)). The opposite, *homonyms*, have the same effect. A homonym is present in code when one identifier is used for multiple concepts. To know which concept is referred you would have to know the context of the usage. This needlessly increases the effort that is needed for understanding (see 2.1(b)). Therefore we define, after Deissenboeck and Pizka [8]:

Definition. *An identifier name is used consistently when a name only refers to one concept and a concept is only referred to by one name.*

2.3 Consistent method naming

A difficult problem in determining whether identifier names are consistently used in an application is defining all the relevant concepts they can refer to. Deissenboeck and Pizka proposed a human expert to create this mapping between used identifier names and concepts that are relevant within an application [8]. This might be a good approach if this is done while the application is being developed but can be a costly task when done on an existing application [17].

Høst and Østvold manage to overcome this problem by not predefining the used concepts of an application. Instead they investigate how the names are used [13].

They made the observation that the intention of a method is manifested in the method body itself. A simple example shows their point.

Listing 2.1: A simple example (originated in [13])

```
public Person _(String name) {
  for (Person p : persons) {
    if (name.equals(p.getName()) {
      return p;
    }
  }

  return null;
}
```

Listing 2.1 shows a method of which the name is omitted. Even without the name it will probably be clear that this method does some sort of *find* operation of a *Person*. Giving this method the name *findPerson* sounds just about right. That we have a notion of which name this method should bear is a pointer that the method radiates information about its intention, so called *implementation clues* [13].

2.4 Finding implementation clues

Høst and Østvold investigate how these implementation clues can be formalised so that they can be automatically recognized [13]. They defined a set of *semantic attributes* which they described as "*properties that a given implementation may or may not possess*" [13]. Semantic attributes should be easy to identify by simple static analysis [12, 26]. An example of such attributes is the prevalence of a loop or the absence of method parameters in a signature. No whole-program analysis should be needed to find these attributes.

The intention of a method can be described with these semantic attributes. By aggregating these semantic attributes in a given method a meta description of a method emerges. This description is of a higher level of abstraction than the method implementation itself. Høst and Østvold dubbed this "*usage semantics*" [13]. The usage semantics of a method is not endless —like the implementation of a method can be— but is always as long as the defined set of semantic attributes. Comparing sets of semantic attributes is therefore a trivial task.

Another term for a semantic attribute is a *nano pattern* [12, 26]. We will use this term in the remainder of this thesis.

2.5 The Nano Pattern Catalogue

Throughout their research Høst and Østvold use different sets of nano patterns [13, 14, 15, 16]. They devised different sets of patterns which they found most fitting for the research task at hand. Although this approach is perfectly viable the problem is that there is no insight in the inter-working of the chosen attributes ("does pattern A mean

Category	Name	Description
Calling	**NoParam**	Takes no argument
	NoReturn	Returns void
	Recursive	Calls itself recursively
	SameName	Calls another method with the same name
	Leaf	Does not issue any method calls
Object-Orientation	**ObjectCreator**	Creates new object
	FieldReader	Reads (static or instance) field values from an object
	FieldWriter	Writes values to (static or instance) field of an object
	TypeManipulator	Uses type casts or instanceof operations
Control Flow	StraightLine	No branches in method body
	Looping	One or more control flow loops in method body
	Exceptions	May throw an unhandled exception
Data Flow	LocalReader	Reads values of local variables on stack frame
	LocalWriter	Writes values of local variables on stack frame
	ArrayCreator	Creates a new array
	ArrayReader	Reads values from an array
	ArrayWriter	Writes values to an array

Table 2.1: nano pattern catalogue as introduced by Singer et al. in [26]. Boldface patterns were originally introduced by Høst and Østvold in [13]

.

that pattern B exists?") or the distribution of the patterns throughout the corpus of code ("how many methods contain this pattern?"). Høst and Østvold are aware of this problem as they note "a more structured approach would be to use the marginal entropy of individual attributes to select from a pool of candidate attributes those that provide the best separation power" [13].

Singer et al. analyse a fixed set of nano patterns in their research [26]. The set they chose is very similar to the set used by Høst and Østvold in their initial research on the used verbs in method names [13]. Singer et al. find that the nano patterns in their catalogue are fairly independent meaning that whether or not a nano pattern is present in a method is not easily predictable [26]. A full overview of the nano pattern catalogue is shown in Table 2.1.

2.6 Method Naming Anomalies

Method Naming Anomalies are method implementations that differ from the intention of the given method name. These are methods which implementations deviate from the common usage of a method name. An simple example can illustrate this.

Lets assume that the method of Listing 2.1 was called *containsPerson*. Given this name we would expect that this method would tell us, a simple yes-or-no question, if the list contains a person with the given name. When we inspect the method body we find that this method does not only tell us that the list contains a person with the given name but it will also return the person object if it is present in the list. Suddenly our

expectation of what this method is supposed to do and what it actually does is in conflict. This would be considered a method naming anomaly.

2.7 Locating Method Naming Anomalies

Høst and Østvold propose to use a "wisdom of the crowds[1]" technique to locate method naming anomalies [14]. They made the observation that in software we do have access to the intended idea of the developer when he named a method: the method body. Høst and Østvold devise a way to find methods that act in contrary with this intended idea. Given enough methods with the same name we can analyse what the commonly used patterns are. If we find methods that act against these commonly used patterns we consider this a method naming anomaly. Based on the definition of Deissenboeck and Pizka [8] we define consistent method naming as:

Definition. *A method name is used consistently if the method name and the the implementation are in line with common patterns (the concept) exhibited in methods that have a similar name.*

Originally Høst and Østvold called a violation of the above definition a *Method Naming Bug* but we propose not to use this term. Instead we propose to use the term *Method Naming Anomaly*. Calling these outliers *bugs* feels to heavy at this point in time since we are not sure whether the found inconsistency are really *bugs* like we have other software bugs [14].

In the remainder of our research we will analyse different ways of finding these method naming anomalies. We will use the method as presented by Høst and Østvold [14] and we will construct our own method of identifying with the use of *Formal Concept Analysis* .

2.8 Related work

The naming of identifiers has been a field of research that has gained some interest in the passed decade. The coming sections give an overview of this research. For convenience reasons we divided it into three different categories: *the role of concepts and names on program comprehension*, *checking for consistent identifier names* and *using patterns for source code analysis*.

2.8.1 The role of concepts and names on program comprehension

Rajlich and Wilde note that concepts play an important part in human learning and that there is much overlap between human learning and program comprehension [23]. To find concepts in code maintainers often resort to searching for identifiers that are used but they note that it has serious deficiencies because there can easily be a mismatch between

[1] "Wisdom of the crowds" means that the collective opinion of a group is taken into account instead of that of one expert (http://en.wikipedia.org/wiki/Wisdom_of_the_crowd)

the term the maintainer uses for a concept and the term the creators of the system used for the concept.

Liblit et al. inspect the cognitive perspective of naming [19]. They state that metaphors are central in naming program entities as they are useful vehicle to represent domain tasks and in making abstractions concrete. In code these metaphors can be referenced by certain names, so called concept keywords.

Lawrie et al. investigate whether use of full English-words in identifiers lead to better program comprehension. By setting up an experiment in which more than hundred and twenty people participated, mostly professionals and some students, they where able to conclude that the use of full English-word identifiers lead to better understanding than abbreviated forms [18].

Caprile and Tonella analyse the structure of identifiers [5]. They devise a grammar which describes the 'language of identifiers'. With this grammar they parse ten C programs and investigate the decomposed identifiers. They envisage that this grammar can be used for forward and reverse engineering tasks.

Deissenboeck and Ratiu construct a meta-model that map names present in an application onto concepts which root from ontologies [9]. This meta-model can be populated using a semi-automatic process. Their main motivation is that using this meta-model it becomes possible to map the low-level source code onto the ontology which describes the high-level intentions and information. They envisage that such a model can be of benefit in detecting naming defects because it makes the relations between the used names in a program and the concepts they represent explicit. Finding synonyms and homonyms for concepts is then an easy task [9].

2.8.2 Checking for consistent identifier names

Deissenboeck and Pizka define definitions for consistent and concise naming [8]. They create a tool that uses this definition to check for identifiers that refer to the same concepts (homonym violations). This tool, which they called Identifier Data Dictionary, finds these homonyms based on the types of identifiers. The tool relied on a human expert to create a mapping between identifiers and concepts.

Lawrie et al. investigate consistent naming by decomposing the identifier names them self [17]. By looking at the structure of an identifier name they found possible synonym identifiers. Their method required the analysis of the syntactical elements of an identifier name and comparing them with each other. Finding identifiers that contain the same syntactical elements are considered a violation. They call this method *syntactic consistency violations*.

Thies and Roth used a slight different approach [27]. In their approach they look at harmonizing identifier names by looking at the variable assignments. They argued that if multiple references exist to a single object they ought to have the same name. This can also be considered a way of finding possible identifier synonyms. If two variables have a different names but reference the same object (concept) this is considered a synonym violation.

2.8.3 Using patterns for source code analysis

Gil and Maman introduce the term *Micro Patterns* which are class level traceable attributes [12]. They define a catalogue of 27 micro patterns and examine the prevalence and inter-relations of the patterns in a corpus of code. They showed that the distribution of the different micro patterns is dependant on the type of application and that the use of micro patterns tend to be the same when analysing consecutive releases of the same application. Notably, it is also Gil and Maman who propose to use the term nano patterns for method level patterns[2].

The correspondence between micro patterns and class names is investigated by Singer and Kirkham [25]. They concentrate on the suffix of the class name, like *Impl* and investigate whether there is a relation between these suffices and the used micro patterns. They use this knowledge to build an ECLIPSE plug in which gives suggestions to the developer on which micro patterns are often included in other classes that contain the same suffix.

As noted earlier, Høst and Østvold have an extended track record when it comes to nano patterns and source code analysis. By analysing the prevalence in methods starting with the same verb they generate a lexicon which they call *The Programmer's Lexicon* [13]. This lexicon describes in natural language the nano patterns that are often used in forty commonly used verbs to start method names with. In [16] Høst and Østvold use nano patterns to find methods that contain similar actions (according to the present nano patterns) but which method names start with different verbs. These methods are considered synonyms of each other. Their goal is to identify all verb synonyms so that they can be eliminated and replaced by a single verb.

[2]They also propose to use the term milli patterns for package level patterns

Chapter 3

Finding Nano Patterns

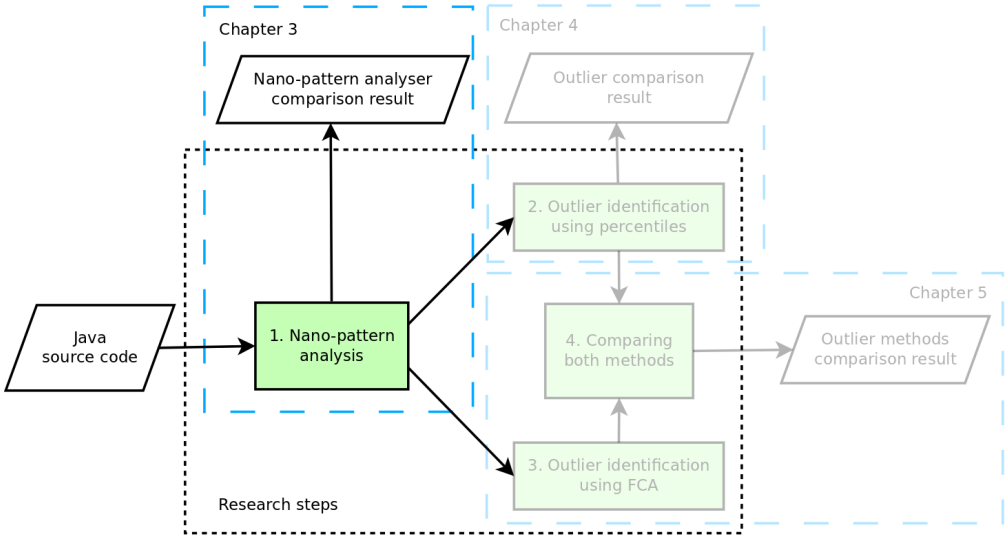


Figure 3.1: Step 1: Locating nano patterns by analysing source code

In the first step of our research we analyse JHotDraw, a Java application, and identify the present nano patterns. We use this data to analyse the difference between the nano pattern byte code analyser that was used by Singer et al. and our created source code nano pattern analyser [26].

3.1 Rationale

Earlier research by Høst and Østvold and Singer et al. searched for nano patterns by analysing the byte code [14, 26]. We propose to use the source code instead.

Information can be lost during compilation like in the case of Java, generic type information or source code annotations. Next to this the compiler can change the structure of the code to for instance enhance the runtime performance of the application.

We believe that using source code instead of byte code might be more suited for this analysis because we want to analyse the actual code written by the developer instead of the interpreted code by the compiler.

Throughout their research Høst and Østvold used different definitions of nano pattern catalogues [13, 14, 15, 16]. They chose the nano patterns as they saw fit for the task at hand. This is a perfectly viable approach but makes it hard to reason about the used nano pattern set itself. For instance, it is not known whether the chosen nano patterns influence each other. Singer et al. do show this for the nano pattern set they chose for their research by using techniques like association mining [1] For this reason we chose this nano pattern catalogue as basis for our research [26].

The main question we will try to answer in this chapter is

Question 1. *How does our source code nano pattern analyser compare to the byte code analyser created by Singer et al.?*

To answer this question we will compare the outcome of both analysers by using some of the same methods Singer et al. used in their research on nano patterns [26].

3.2 Research method

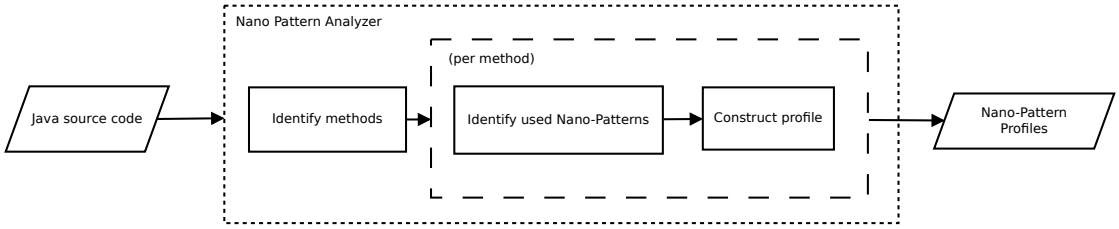


Figure 3.2: Overview of the different steps to identify the nano patterns

Figure 3.2 shows the process of locating nano patterns. In this step we focus on analysing the method implementation.

3.2.1 Analysing the method implementation

Analysis tool

We use Rascal[1] as our main analysis tool. Rascal is a meta-programming environment developed at CWI. It has built-in features for software analysis purposes like querying and manipulating the Abstract Syntax Trees (AST) of, for instance, Java code. We use this feature for locating nano patterns.

[1] http://www.rascal-mpl.org/

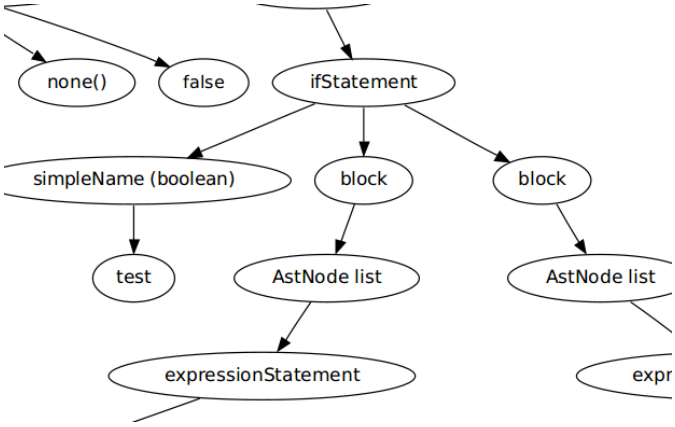


Figure 3.3: An example of Java AST created by Rascal

Analysing the Abstract Syntax Tree

The AST contains nodes for every syntactic construct in the source code. Different operations have different types of nodes. For instance, when an if-else construct is used in the source the AST will contain a node *ifStatement* with a *booleanExpression*, a *thenStatement* and an optional *elseStatement* as children. Each child can have children of their own. To check for the presence of nano patterns we check for the presence of certain nodes, or combinations of nodes in the AST. The `Object Creator` nano pattern for instance can be checked by looking for the presence of the `Class Instance Creation` node in the AST. The definition of the nano patterns in terms of AST nodes is described in appendix B.

In the current implementation of Rascal the Eclipse Java Development Tools (JDT) is used to construct the AST. Therefore the structure of the AST in Rascal has strong resemblance with the AST structure as constructed by JDT. We have defined 80 possible nodes, syntactic constructs, that can be present in the AST. Appendix A shows an overview of all the different nodes.

Constructing nano pattern profiles

The result of the localizing nano pattern process is stored in a nano pattern profile. A nano pattern profile describes the existence of individual nano patterns in a method (see Figure 3.4). A profile can be seen as a meta-description of a method; a fixed length and binary description. Profiles function as a base for comparing methods with each other.

3.2.2 Qualifying the result

To compare our source code analyser with the byte code analyser used by Singer et al. we use two methods that were also originally used by them [26]. The first method is comparing the frequencies of the different nano patterns found by the source code and

	NoParam	NoReturn	Recursive	SameName	Leaf	ObjectCreator	FieldReader	FieldWriter	TypeManipulator	StraightLine	Looping	Exceptions	LocalReader	LocalWriter	ArrayCreator	ArrayReader	ArrayWriter
getOriginalText(), action-token: ***get***	**1**	0	0	0	**1**	0	**1**	0	0	**1**	0	0	0	0	0	0	0
createImagesMenu(), action-token: ***create***	**1**	0	0	0	0	**1**	**1**	0	0	0	**1**	0	**1**	**1**	0	**1**	0

Figure 3.4: Examples of a nano pattern profiles taken from JHotDraw

byte code analyser. The second method is comparing the entropy score of the result found by both analysers.

The byte code analyser of Singer et al.[26]

We do not make use of the pattern frequency table as described by Singer et al. because this would mean we have to exactly mimic their earlier research with the same corpus of code [26]. Instead we use the byte code analyser implementation of Singer et al. which was made available by them.[2]

Nano pattern frequency

Calculating the frequency of a nano pattern is straightforward. The frequency is calculated per pattern. We count the times a pattern is present in the analysed methods and divide this by the total amount of analysed methods. We do this for all 17 patterns in our catalogue.

Shannon's entropy; the predictability of nano patterns

In information theory entropy relates to the uncertainty of a random variable[3]. In other words entropy measures the predictability of a variable. For instance, a coin toss with a fair coin is unpredictable, you can not predict whether it will end up heads or tail. On the other hand tossing a coin with heads on both sides has no uncertainty, it will end up heads on every toss.

Singer et al. uses entropy to calculate the predictability of the nano pattern catalogue [26]. The lower the score the more predictable the presence of nano patterns in a code base are. Having a low score would mean that nano patterns are not that useful for classification of source code; the existence of the patterns in all code would be very predictable and so they can not be used to differentiate between different pieces of code like different methods [26].

[2]The binary and source code of the nano pattern byte code analyser by Singer et al. was downloaded from http://www.dcs.gla.ac.uk/ jsinger/code.html

[3]http://en.wikipedia.org/wiki/Entropy_(information_theory) on 21-03-2012

To measure the entropy score Singer et al. use the definition of Shannon's entropy (see Definition 3.1).

$$H = -\sum_{b \in B} p_b \log_2(p_b) \tag{3.1}$$

B contains all possible nano pattern profiles. There are 17 nano patterns in our catalogue. Because of the binary nature of nano patterns the total set of possible nano pattern profiles B consists of 2^{17} elements. Therefore the highest number of entropy that the nano patterns can exhibit is $\log_2 |B|$ which is 17. p_b is defined as the probability of the occurrence of a certain nano pattern profile $b \in B$ in the total corpus (see section 4.2.1) [26].

To compare the byte code analyser of Singer et al. with our source code analyser we will calculate the entropy score on the outcome of both methods. If the score has risen the uncertainty of the nano patterns have increased. If it has lowered the uncertainty decreased.

3.3 Results

3.3.1 Difference per pattern

Table 3.1 shows the difference per pattern between of the byte code analyser used by Singer et al. [26] and the source code analyser created by us. There are several reasons why these differences occur. We have highlighted some of the most significant differences in the results.

3.3.2 Number of analysed methods

As is shown in Table 3.1 there is a difference in the number of analysed methods. This is due to anonymous inner classes that are created in the constructor of a class. When an anonymous inner class is created it will be compiled into a second class[4] containing

- an initializer
- the overridden or added methods

The byte code analyser will analyse this created class like it does every other compiled class file. The initializer method will be skipped but the overridden or added methods will be analysed. These methods are skipped by the source code analyser because they are defined inside a constructor which is skipped completely. This results in 21 methods that are analysed by the byte code analyser but missed by our source code analyser.

[4] which has the same name as the containing class suffixed by $[nr]

nano pattern	Byte code analyser (in %)	Source code analyser (in %)	Difference (in %)
NoParam	64.8	64.8	0.0
NoReturn	67.2	67.2	0.0
Recursive	0.3	0.3	0.0
SameName	20.9	21.0	-0.1
Leaf	41.6	47.6	-6.0
ObjectCreator	19.6	19.2	0.4
FieldReader	26.8	21.1	5.7
FieldWriter	17.0	16.4	0.6
TypeManipulator	7.1	8.4	-1.3
StraightLine	80.7	84.5	-3.8
Looping	6.0	6.0	0.0
Exceptions	40.9	41.1	-0.2
LocalReader	64.5	36.6	27.9
LocalWriter	17.9	17.8	0.1
ArrayCreator	1.5	0.7	0.8
ArrayReader	2.2	2.2	0.0
ArrayWriter	1.4	1.4	0.0
Total # of methods	**4407**	**4386**	**21**

Table 3.1: Coverage scores for each nano pattern as present in JHotDraw (version 709) found by the byte code analyser and the source code analyser

3.3.3 Differences in pattern presence

Local Reader

The presence of the `Local Reader` pattern differs greatly, 27.9%, between the two analyser implementations. This is by design. The original description given by Singer et al. of the `Local Reader` pattern was "reads values of local variables on stack frame" [26]. In the implementation of the byte code analyser the pattern is assigned to a method when one of the `load` operation codes[5] is present in the method. These are very prevalent operation codes. For instance if an operation needs the current object, `this`, because it calls another method or uses the value of a class variable, the current object will get loaded to the local stack frame resulting in the occurrence of the `ALOAD` operation code [2].

The implementation of the `Local Reader` pattern in the source code analyser is different. We interpreted `Local Reader` as "reads a variable which is defined in the scope of a method". The source code analyser will therefore only trigger the presence of the `Local Reader` pattern when a local method variable is read, not when a variable gets loaded onto the local stack frame. By using this definition the `Local Reader` and `Field Reader` patterns are natural counterparts of each other. `Local Reader` is only triggered for variables that are defined within the scope of a method whereas `Field Reader` is

[5]The possible load operation codes are: ILOAD, LLOAD, FLOAD, DLOAD, ALOAD

only triggered for variables that are defined in the scope of a class.

Leaf

The number of methods in which the `Leaf` pattern is present is 6% higher when the project is analysed by the source code analyser compared to the byte code analyser. This difference can be explained by the difference in object creation between byte code and source code. In byte code the operation code `new` (object creation) will be followed by the `invokespecial` (method call) code which will invoke the initializer, the constructor, of the newly created object. This `invokespecial` is seen by the byte code analyser as a method call and thus breaking the rule of the leaf pattern. The source code analyser does not contain the same implementation of the leaf pattern. To abide to the leaf pattern rule a method should not call any other method. The implementation of the source code analyser therefore only looks for other method or super method invocations in a method. A class instantiation by itself is not seen as a method call. If no method is invoked on the newly created object then no method call is registered by the source code analyser. Listing 3.1 contains an example of the difference.

Listing 3.1: Listing of a method that complies to the *Leaf* pattern in the source code analyser but breaks the pattern according to the byte code analyser

```
public List createList() {
  return new ArrayList();
}
```

3.3.4 Difference in entropy score

Like Singer et al. we have calculated the entropy score (after Shannon) on JHotDraw for both the byte code analyser and the source code analyser (see Table 3.2). The entropy score measured on the source code analyser is slightly higher than the score measured on the byte code analyser. This is probably mostly due to the different implementation of the `Local Reader` pattern.

Entropy measures the uncertainty of a random variable. The uncertainty of the entire nano pattern catalogue has gone up. In the byte code analyser the presence of the `Field Reader` pattern did often imply the presence of the `Local Reader` pattern. The source code analyser handles this differently. Here the existence of the `Field Reader` pattern does not imply the existence of the `Local Reader` pattern (see section 3.3.3). Therefore the separation power of the nano pattern catalogue goes up.

3.4 Analysis

Both byte code and source code seem to give similar results. The biggest difference is the difference of the `Local Reader` pattern. This can be explained because we used another interpretation of the pattern Other patterns also show some different prevalence scores.

	Byte code analyser	Source code analyser
Total # of methods	4407	4386
# of unique pattern profiles	321	453
Entropy score	**5.43**	**5.78**

Table 3.2: Difference in entropy score (after Shannon) between the byte code analyser and the source code analyser measured in JHotDraw (version 709)

Most of these differences can be explained by the slight differences between the source code and the byte code or a slight interpretation difference.

All-in-all we conclude that our source code pattern analyser is a good basis for our further research since it shows similar results than the byte code analyser of Singer et al., both in pattern presence as in entropy score. This makes us conclude that the patterns found by our source code analyser will have similar or better separation powers as the patterns found by the byte code analyser.

Next to the above we envisage that the source code analyser can be extended in the future with patterns that can easily be identified in the source but might be problematic when analysing the byte code. This is because the source code analyser has access to the AST which contains information that might be lost when only analysing the byte code. An example is generic type information[6]. The source code analyser still has access to this information while the byte code analyser does not since this information is lost during compilation[7].

3.5 Conclusion

In this chapter we wanted to get an answer to the question:

Question 1. *How does our source code nano pattern analyser compare to the byte code analyser created by Singer et al.?*

Both analysers show similar results. The difference can be explained by different interpretations of the meaning of a nano pattern and some slight differences in structure between the source code and the compiled byte code. The entropy score has slightly gone up which indicate that the separation power of the whole nano pattern catalogue has slightly increased.

[6]Generic type information was added to the Java language in version 5

[7]It is possible to access this information via the ASM library but it then makes use of the Java reflection API and not of the byte code [2]

Chapter 4

Finding method naming anomalies using the method of Høst and Østvold

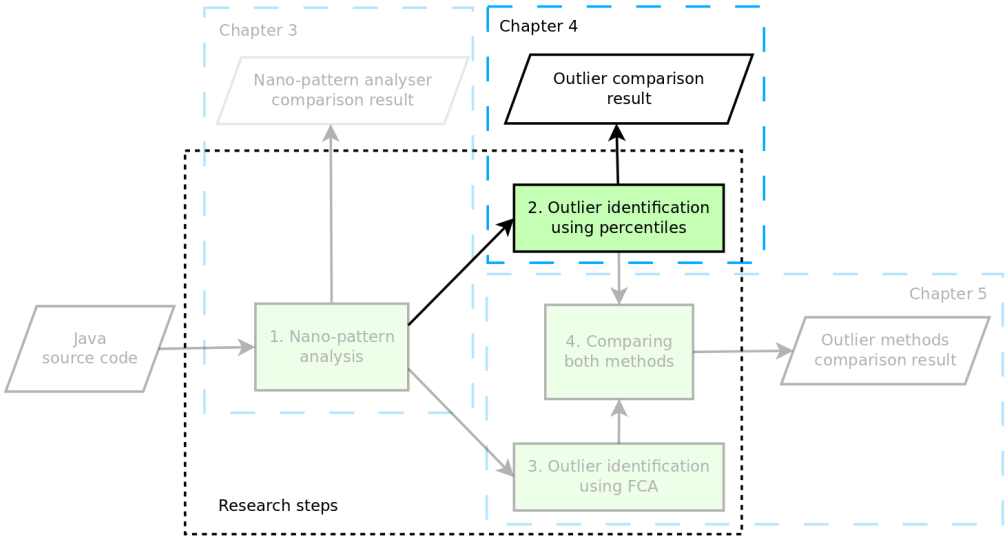


Figure 4.1: Step 2: Finding method naming anomalies using the method of Høst and Østvold

4.1 Rationale

In this step we use the method of Høst and Østvold to identify possible method naming anomalies [14]. By reproducing their steps and analysing the outcome we try to get more insight in the nature of method naming anomalies which are found using their method.

To summarize, we will try to answer the following question:

Question 2. *Do methods that get identified by this method have bad names?*

4.2 Research method

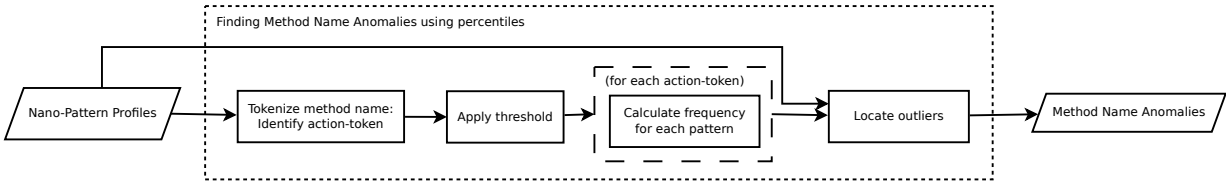


Figure 4.2: Overview of the different steps to identify method naming anomalies using the method of Høst and Østvold

In this section we will explain the different steps we take to find method naming anomalies. By applying the steps shown in Figure 4.2 we will locate methods with method naming anomalies. We can draw conclusions by judging whether these found outliers, methods with method naming anomalies, truly have bad names.

4.2.1 The Java corpus

The corpus that we use for finding method naming anomalies contains a cross section of projects used in earlier research by Høst and Østvold and Singer et al. By choosing the same projects we want to minimize the chance that different projects give different result than earlier research has shown. Table 4.1 gives an overview of all the different projects in our corpus of code.

The research of Høst and Østvold divided the projects in the corpus into different software categories [14]. We follow this approach.

In our corpus each category contains between 14.000 and 23.000 methods[1]. Although the categories are somewhat arbitrary chosen, we do agree with Høst and Østvold that it is good idea to take the domain of the application into account to prevent the possibility that the corpus might be skewed towards a certain domain. If the corpus would lean heavily on a certain domain it could potentially reduce the applicability of the found results.

4.2.2 Analysing the method name

The structure of method names

Although a method name can be any random combination of characters this is most often not the case. In most cases the method name is a structured string and is constructed using a combination of verbs, nouns, adverbs and adjectives [5]. According to the original

[1] with the exception of the Jakarta Commons Utilities category which only contains 8.000 methods because there are simply not enough Jakarta Commons projects

Category	Name	Version	# Analysed methods
Benchmarking	JikesRVM*	2.9.1	13958
Desktop application	JHotDraw*	709	2466
	JEdit	4.3	5820
	ArgoUML	0.24	9038
Jakarta common utilities	Commons Collections	3.2	3210
	Commons Net	1.4.1	927
	Commons Digester	1.8	420
	Commons Codec	1.3	170
	Commons Lang	2.3	1785
	Commons HTTP-client	1.0.1	1164
	Commons IO	1.3.1	431
Language and language tools	BSF	2.4.0	294
	ASM	2.2.3	918
	BCel	5.2	2043
	JRuby	0.9.2	5230
	Polyglot	2.1.0	3993
	Antlr	2.7.6	2066
Middleware, frameworks and toolkits	Struts	2.0.1	2267
	TranQL	1.3	1323
	Tapestry	4.0.2	3037
	Spring	2.0.2	9122
Programmer tools	Ant	1.7.0	7747
	JUnit	4.2	569
	Velocity	1.4	1366
	FitNesse**	2011-01-04	6868
Server and database	JBoss***	3.2.7	19125
Utilities and libraries	XML-Batik	1.6	9328
	Hibernate	3.2.1	9739
	Ognl	2.6.9	713
	JarJar Links	0.7	244
XML tools	Castor	1.1	7808
	Xalan-J	2.7.0	8479
	Xerces-J	2.9.0	6494
		Total	**148162**

Table 4.1: Overview of the analysed corpus.
(*) Analysed by Singer et al. [26] not by Høst and Østvold.
(**) Analysed version by Høst and Østvold unkown [14].
(***) Analysed version differs from the version analysed by Høst and Østvold [14]

.

Sun Java Code Conventions method names should be verbs like `run()`, `runFast()` or `getBackground()` [22]. This convention is well established and known in the JAVA development community but there are cases where non-verbs are used at the beginning of method names. For some actions, like transformations, it is known that other words like 'to' are used as the first token of a method name [5]. Other possibilities are the so called *indirect actions* on objects like the method `size()` on lists where the intention is to get the size of the list but because this is such a well-known operation the verb is omitted [5].

Because of these exceptions we will refer to the first token of a method name with the term *action-token*.

Tokenizing a method name

To identify the action-token we use a simple token capitalisation strategy. We split the method name on the first occurrence of upper case letter and use the first token as the verb of the method. For example the identified action-token of the method `getName()` would be `get`.

Again this strategy is in line with the original Sun Java Code Conventions which states that method names should be "... mixed case with the first letter lower case, with the first letter of each internal word capitalized" [22] . Butler et al. show that 91.6% of the method names in a large JAVA corpus are indeed composed in this manner [4]. By using this strategy we should be able to correctly identify the first token of a method name in approximately nine out of ten times. By applying a threshold (as described in section 4.2.4) we suspect that we will flush out any remaining incorrect tokenized names.

4.2.3 Grouping methods per action-token

We calculate the frequency of nano patterns per action-token. To do this we group all the methods that start with the same action-token. For instance, all the methods that start with the token `accept` are considered as one set, a so called *action-token set*. By doing this we get insight in which patterns are often or rarely used in methods starting with the same action-token (see Figure 4.3).

4.2.4 Method occurrences threshold

Not all methods are used in our analysis. If we want to be able to compare methods with each other which share the same action-token there should be a certain number of these methods available. Trying to find a common pattern for an action-token does not work if there are only two method instances available starting with this token. We should apply a threshold to make sure that a specific token has a minimal occurrence.

Next to the above problem we also have a potential other problem. It can be that a certain action-token is only used in a specific application or in a specific application domain. This would mean that the found common patterns would only be applicable for that application or application domain. This is in violation with the idea that we want to tap into the common understanding of developers community and not into the

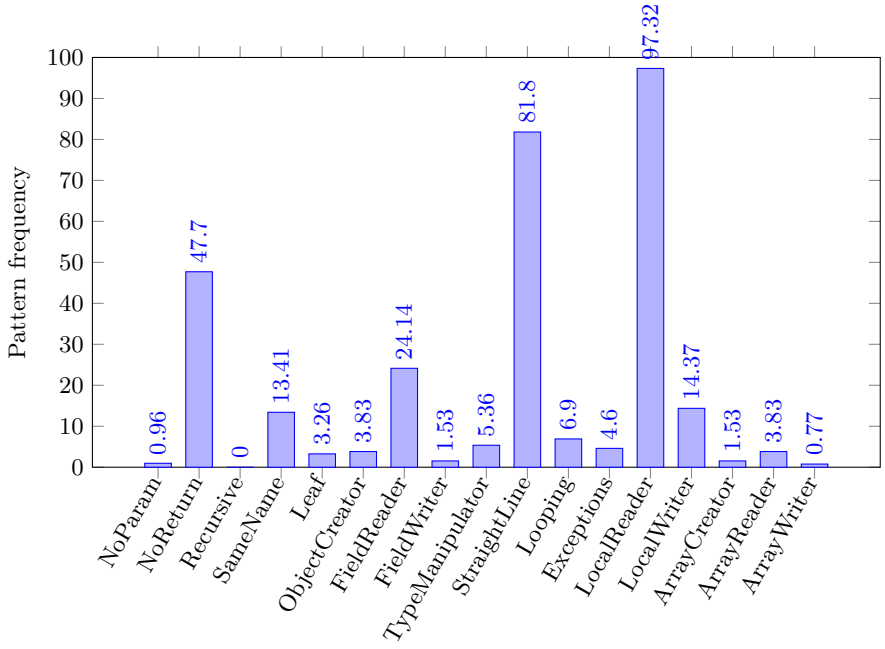


Figure 4.3: Frequency of the different nano-patterns for methods that start with *accept* (calculated over 522 methods)

understanding of the developers of a specific application. To prevent this we should also apply a threshold on the number of projects in which a certain action-token needs to occur.

In their research Høst and Østvold also apply a threshold [14]. They state that a method name should be present in at least half of the projects and should have a minimum occurrence of 100 times. Their total corpus contains more then 1.000.000 methods making 100 methods about 0,01% of the corpus. If we apply the same kind of relative threshold value to our corpus it would mean that an action-token should occur 15 times (0,01% of the total amount of methods in our corpus) and should be present in at least 16 projects. The need for a method to be present in at least 16 projects overrules the other threshold value therefore we will only apply the project threshold value.

4.2.5 Finding naming anomalies using nano pattern frequencies

Høst and Østvold observed that the nano pattern frequencies each action-token set exhibits could be used to find outliers [14]. For instance, if we find a method that includes a nano pattern that is not often used in other methods within the same action-token set this could be considered an anomaly. Figure 4.3 shows the nano pattern frequencies of all the methods that start with the action-token `accept`. None of the methods in this action-token set do a recursive method call.[2] Finding a method which starts with the

[2]The nano pattern "Recursive" has a frequency value of 0 for the action-token set `accept`

action-token `accept` that does do a recursive method call would mean that it is the first method of its group that would exhibits this behaviour. Because this behaviour is so rare for this action-token set we would therefore considered this method as an outlier.

The attribute rule set

To determine whether or not a method is an outlier Høst and Østvold devised a set of rules they called the *attribute rule set* (see Table 4.2). Each rule corresponds to a partition of the frequency set. When a method is in violation with one of the `inappropriate` rules it is considered an outlier. For instance, if a method does not contain a nano pattern which is present in 98% of the other methods of its action-token set, it violates the `inappropriate if omitted` rule and thus making the method an outlier.

The rule set of Høst and Østvold distinguished between three different levels of severity of violations [14]. In our research we do not make this distinction. The question what the severity of a found outlier is might be interesting but at this moment a little premature. First we have to determine whether the found outliers are truly outliers. Then we can focus on gradations of severity of the found outliers. To put it in other words, first we have to determine whether the found outliers are indeed methods that have a bad name, then we can experiment whether or not there is a gradation of badness regarding naming.

We propose to use an attribute rule set consisting of 3 rules. These rules are `inappropriate if included`, `no violation` and `inappropriate if omitted`.

	Original attribute rule set			Narrowed attribute rule set		
Severity	From	-	To	From	-	To
Inappropriate if included	0.0	-	5.0	0.0	-	3.0
No violations	5.0	-	95.0	3.0	-	97.0
Inappropriate if omitted	95.0	-	100.0	97.0	-	100.0

Table 4.2: Attribute rule sets using different frequency partitions

The rule `no violation` indicates that there is no violation if a method includes or omits the pattern. For instance, when we again take a look at the frequency set of `accept` (Figure 4.3) this would mean that the presence or absence of the `No Return`, `Same Name`, `Field Reader`, `Type Manipulator`, `Straight Line`, `Looping` and `Local Writer` patterns in a method starting with `accept` would not trigger any violations.

The narrowed attribute rule set

The original values of the attribute rule set as determined by Høst and Østvold were arbitrary chosen [14]. To investigate what the influence of these values are we propose the use of a second attribute rule set. This rule set has narrowed partitions (see Table 4.2). This means that the range of frequencies in which the occurrence or omission of a nano pattern in a methods is considered a violation is narrowed. We hypothesise that by

choosing a narrowed range we will find less outliers but these outliers will more often have a bad name.

4.2.6 Qualifying the result

Qualifying the threshold mechanism

By applying the threshold we will remove methods from our corpus. We will investigate the lost methods by manually inspecting the 'drop-out' list and inspecting the different reasons why these methods were excluded. We will also take a look at the list of the methods that are included.

Qualifying the difference between the two attribute rule sets

By comparing the statistics for the found outliers for both sets and investigating the noticeable differences we get more insight into the influence of the values of the attributes set.

Qualifying the identified anomalies

In their original research Høst and Østvold investigated 50 randomly chosen outliers to judge whether or not the found outliers were truly 'buggy' method names. To do this they relied on their own best judgement.

We will use a similar approach with the addition of a control group. We will take a random sample from the outlier list and mix this with a random sample from the total analysed method list. When we judge the samples on the list it is not known to us whether the chosen sample was marked as an anomaly or not. The samples from both groups, anomalies and non-anomalies, are randomly distributed in our sample list. We make sure that the samples that are taken from the total analysed method list are not on the outlier list. Of each outlier we will motivate whether or not it has a badly chosen name. To do this we will consult the method and class implementation and, if present, the JavaDoc.

By using this sampling technique we want to minimize the effect of our own personal bias towards the found outliers. By mixing the samples that are and are not marked as outliers we can see whether our judgement on the correctness of the used names is in-line with the results returned by the outlier method. It could be for example that we would identify as much naming anomalies in a set of methods that was not marked as an outlier than we would in a set that is marked as outliers. This would reduce the usefulness of this method of identifying anomalies.

4.3 Results

4.3.1 Applying the threshold

As discussed in section 4.2.4 we applied a threshold to filter out all the action-tokens with a low occurrence. Table 4.3 shows the number of methods which were included after applying the threshold.

	Total (in #)	% of total
Methods included after applying the threshold	109730	74.00
Methods excluded by applying the threshold	38563	26.00
Unique action-tokens included after applying the threshold	91	3.27
Unique action-tokens excluded by applying the threshold	2688	96.73
Total # of methods in corpus	148293	
Total # of unique action-tokens in corpus	2779	

Table 4.3: Number of methods and action-tokens in- and excluded after applying the threshold

Out of the analysed methods 74% of the used action-tokens have occurrences above the threshold. This means that 26% of the methods are excluded by applying the threshold. When we look at the action-tokens we see that there are a total of 2779 uniquely used in our corpus. The methods that were excluded because of the threshold contain 2688 different action-tokens. This is 97% of the total action-tokens.

If we investigate the words that were excluded we see a combination of verbs (like 'abbreviate' or 'operate'), nouns (like 'frozen' or 'garbage'), abbreviation (like 'abs' or 'jvm') and random combination of letters (like 'aaa' or 'jj'). In most cases the reason why these words are not included is because they only occur a few times in one or two projects. There are 1621 methods that get excluded because of an unexpected method name. These methods start with an upper-case or an underscore character. In these cases the action-token that is identified by our tokenization strategy is an empty string.

When we take a look at the action-tokens that are most used we see that `get` is by far the most used verb followed by `set` (see Table 4.4). This means that almost 50% of the methods that are included in our research begin with the verbs `get` or `set`. This is to be expected since using `get` and `set` is such a prevalent pattern in Java to manipulate object properties. Other research show similar results on the use of `get` and `set` [13].

Further investigation of the included word list shows that there are non-verbs included (for instance, in the top ten the word 'to' is included). When we analyse the words using WordNet[3] we find that out of the 91 included action-tokens 78 (86%) are actual verbs.

4.3.2 Analysing the found outliers

We have run the outlier method for both the original and the narrowed attribute rule set. Table 4.4 shows an overview of the found outliers. In the coming sections we will

[3]WordNet is a database for the English language. See http://wordnet.princeton.edu/

	# occurrences
get	37847
set	16472
is	6265
add	4106
create	4077
to	3037
visit	1925
remove	1915
has	1514
read	1121
Total	78279

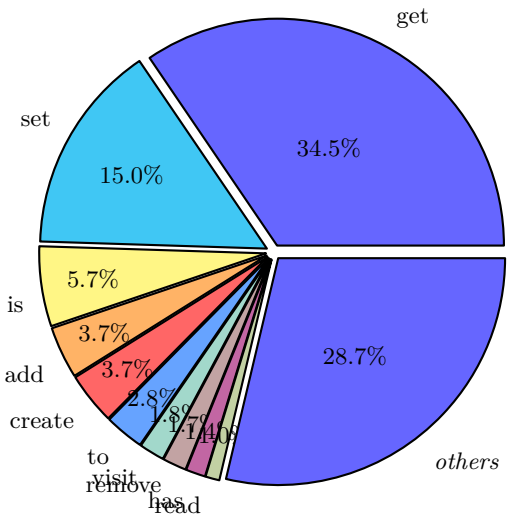


Figure 4.4: Overview of the top ten used verbs in the corpus

investigate the result and take a deeper dive into the found outliers.

Overview of the outliers found with the original attribute rule set

By using the attribute rule set also used by Høst and Østvold we find that out of all the methods in our corpus 10558 (9.6 percent) get marked as anomalies. The Ognl project has the highest reported number of outliers, 24,9%. When we investigate the reported outliers for Ognl we see that out of the 117 reported outliers, 63 are found for the `get` action-token. Almost all these outliers, 62, share the same reason; the methods violate the `inappropriate if included: Array Reader` rule. Returning values from an array seems to be a common pattern used in the Ognl code base.

39 out of the 62 methods share the same name: `getValueBody`. An inspection of the methods show that they all have a similar method structure (see Listing 4.1). As the class names suggest these classes represent an AST and the `getValueBody` methods perform the node operation and return the values. These methods and their accompanying names make sense within the scope of the Ognl application. We would therefore deem these methods as correctly named.

Overview of the outliers found with the narrowed attribute rule set

When we narrow the attribute rule set the number of found outliers go down by 68% from a total of 9111 to 3160 methods. The earlier mentioned outliers found in Ognl for the `get` action-token are now not found any more. An examination of the pattern frequency set for the `get` action-token explains this behaviour; the `Array Reader` pattern is present in 4.3% of all `get`s. Since the narrowed attribute rule set only marks a method in violation if a nano pattern has a occurrence of less then 3%, the presence of this

	(a) Original attribute rule set		(b) Narrowed attribute rule set		(c) Difference		(d)
Project	# methods	% of total	# methods	% of total	# methods	%	%
Ant	475	6.90	193	2.80	282	59.37	0.85
Antlr	77	7.91	23	2.36	54	70.13	1.18
ArgoUML	677	9.58	185	2.62	492	72.67	0.85
ASM	120	14.27	36	4.28	84	70.00	0.30
BCel	168	9.10	75	4.06	93	55.36	-
BSF	22	11.52	7	3.66	15	68.18	0.00
Castor	526	8.06	228	3.49	298	56.65	0.88
Commons Codec	12	8.45	6	4.23	6	50.00	0.00
Commons Collections	271	11.10	71	2.91	200	73.80	1.14
Commons Digester	25	6.67	8	2.13	17	68.00	0.34
Commons HTTP-client	70	6.74	14	1.35	56	80.00	1.46
Commons IO	39	11.02	12	3.39	27	69.23	5.17
Commons Lang	117	8.36	29	2.07	88	75.21	1.93
Commons Net	60	10.58	19	3.35	41	68.33	1.58
FitNesse	481	13.70	106	3.02	375	77.96	2.14
Hibernate	668	8.73	154	2.01	514	76.95	2.00
JarJar Links	25	12.14	1	0.49	24	96.00	0.72
JBoss	1421	9.26	467	3.04	954	67.14	0.95
JEdit	473	12.35	164	4.28	309	65.33	1.30
JHotDraw	61	3.85	16	1.01	45	73.77	-
JikesRVM	629	8.51	160	2.17	469	74.56	-
JRuby	351	10.87	96	2.97	255	72.65	1.27
JUnit	34	11.00	13	4.21	21	61.76	3.50
Ognl	117	24.95	26	5.54	91	77.78	0.45
Polyglot	86	5.48	22	1.40	64	74.42	1.64
Spring	455	5.93	138	1.80	317	69.67	1.52
Struts	152	7.42	13	0.63	139	91.45	1.06
Tapestry	107	4.63	17	0.74	90	84.11	0.87
TranQL	86	7.08	35	2.88	51	59.30	1.17
Velocity	59	6.98	17	2.01	42	71.19	0.67
Xalan-J	834	12.45	345	5.15	489	58.63	1.21
Xerces-J	650	11.99	255	4.70	395	60.77	0.19
XML-Batik	643	8.28	209	2.69	434	67.50	0.76
Total	**9991**	**9.11**	**3160**	**2.88**			

Table 4.4: Violations found with both the original (a) and the narrowed attribute rule sets (b) and an overview of the original presented figures by Høst and Østvold in their research [14] (d)

pattern is not considered a violation any more. In this case this seems just since we would also judge them as correctly named.

By narrowing the partitions all the projects see the outliers reduced by at least 50%. A project that has the most drastic decline of outliers is the JarJar Links project. Here the number of outliers go down from 25 to 1. A comparison of the found outliers with the original and the narrowed attribute sets show that using the original attribute rule set it marked 24 `visit` methods as outliers because they violated the rule `inappropriate if included: No Param`. When we inspect these methods we see that they are implementations of an external interface (from the ASM library) meaning that the name can not be changed by the implementing class (see Listing 4.2). Since this is outside the scope of the JarJar Links project we would judge these methods as correctly named. This judgement is debatable. Having methods called `visit` without parameters that get passed in is in some contrast with the *Visitor* design pattern [11].

Comparing the results with the results of Høst and Østvold [14]

If we do a comparison with the original values reported by Høst and Østvold we see big differences in the reported number of outliers. For some projects, for instance the BSF project, Høst and Østvold do not find any outliers while our method, especially when we

Listing 4.1: Listing of a method from OGNL that was marked as an outlier using the original attribute rule set (ognl.ASTShiftRight)

```
protected Object getValueBody(OgnlContext context, Object source) throws
    OgnlException {
  Object v1 = children[0].getValue( context, source );
  Object v2 = children[1].getValue( context, source );

  return OgnlOps.shiftRight( v1, v2 );
}
```

Listing 4.2: Visit method from JARJAR LINKS that was marked as an outlier using the original attribute rule set but not using the narrowed attribute rule set

```
public SignatureVisitor visitInterface() {
  sw.visitInterface();
  return this;
}
```

use the original attribute rule set, marks more than 11% of the methods as outliers. This could be explained because of three differences between the method we have used and the method used by Høst and Østvold.

1. Høst and Østvold analysed a corpus containing more then a 1.000.000 methods, ours contains 100.000
2. Høst and Østvold analyse the complete method name —the method phrase [15]— we analyse the first token of the name
3. Høst and Østvold used a different nano pattern catalogue

It could be that the number of methods in the corpus influence the constructed frequency set for each action-token. For instance, if there are more methods beginning with the token `add` it could be that there is a different distribution of present nano patterns. In our case we see that the pattern `Local Reader` is present in methods that start with `add` in 95.88% of the cases (see Figure C.4). Having a bigger corpus and thus having more methods that start with the action-token `add` could mean that the presence of the `Local Reader` pattern falls under the boundary of the attribute rule set.

Another difference is that Høst and Østvold analysed the whole method name. They tagged the complete name using a part-of-speech tagger and constructed different frequency sets for the complete method phrase. It could be that the found frequency sets are less distinct than the ones we have constructed for the action-tokens. For instance, almost all of our action-token frequency sets contain at least one pattern were its presence of absence would trigger a violations. This means that our corpus will almost always contain outliers for a given action-token set. Whether or not this is also the case for the

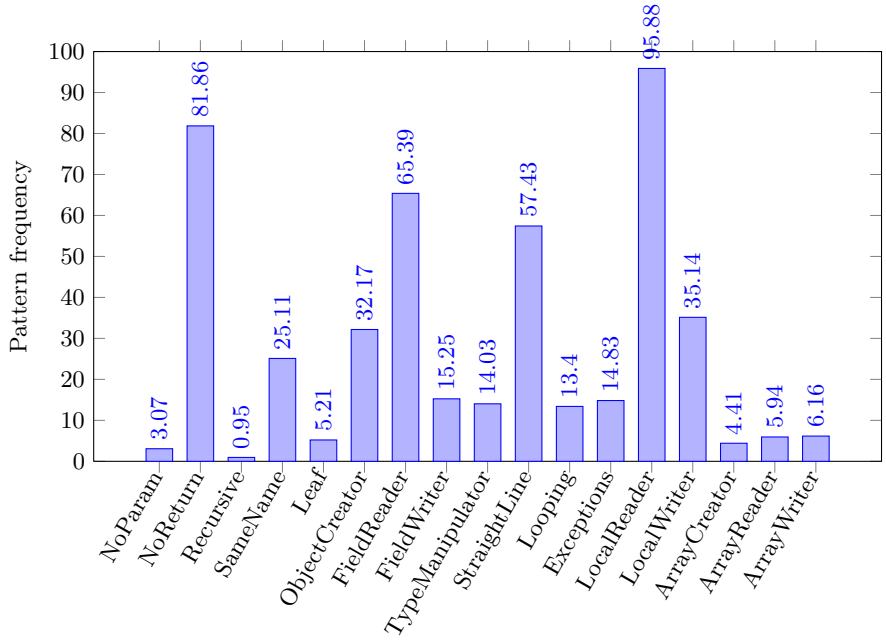


Figure 4.5: Frequency of the different nano-patterns for methods that start with *add* (calculated over 4106 methods)

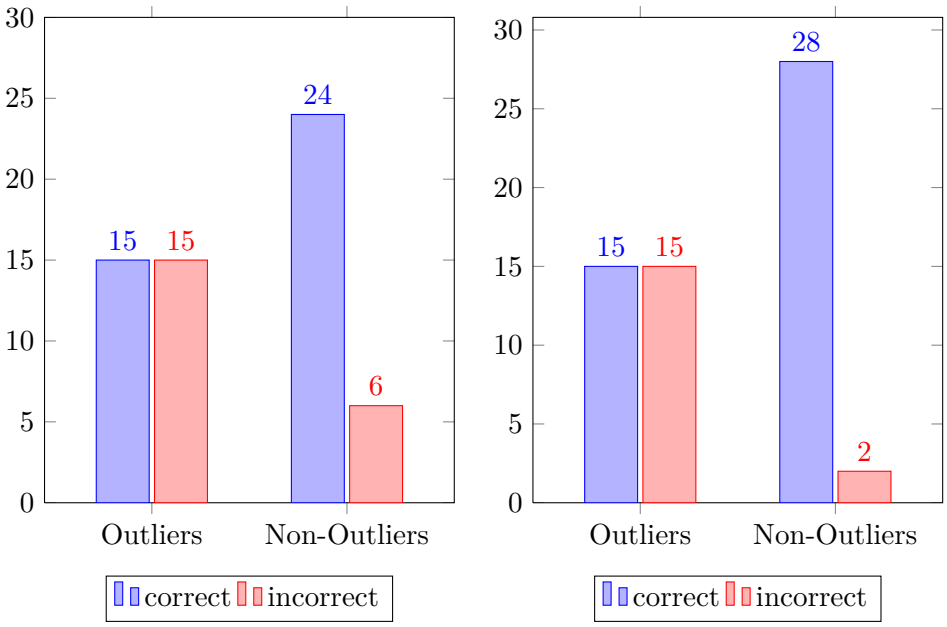
frequency sets that where constructed by Høst and Østvold is unknown, no results have been published on this. Next to this it is unknown how many methods where included after Høst and Østvold applied their threshold. It could be that a lot of methods where lost because of it. In comparison with our method we lost more than a quarter of our initial methods by applying our threshold and we only concentrate on the first token of the name, not on the name as a whole.

The used nano pattern catalogue also differs. This also could have an influence on the number of found outliers. For instance, Høst and Østvold make a difference in return types [14]. Their patterns check whether or not the return type is void, int, string, a reference, a boolean or whether the return type is used in the method name. These patterns are clearly more fine grained than our `No Return` pattern. It could be that the presence of these patterns are more scattered over a certain method phrase rendering these patterns less useful for outlier spotting.

Overview of the judgement review

For each attribute rule set we have judged a set of 60 samples. These samples were taken randomly from both the list containing the method outliers as from the list containing the other methods (see section 4.2.6). During the judgement of each separate method on the list it was unknown to us whether or not it was marked as an outlier.

Figure 4.6 shows the result of our judgement. For both attribute rule sets we argued that 15 of the 30 methods on the outlier list had incorrect names. For the non-outlier



(a) Original attribute rule set

(b) Narrowed attribute rule set

Figure 4.6: Correctly and incorrectly identified naming outliers

list this was lower for both judgement sessions, respectively 6 and 2 times. Appendix D contains an overview of all the samples and accompanying reasoning why we judged whether or not a method had a correct name.

An example of an identified outlier which we consider valid is shown in Listing 4.3. This outlier violates two rules; `inappropriate if included: Array Writer` and `inappropriate if included: No Return`. The method bears the name `getChars` but does not return anything (as was highlighted by the outlier method). Instead it copies a sequence of characters of an internal string into a passed in character array. We would argue that the name `copyChars` would be more appropriate.

Listing 4.3: Listing of a method from Xalan-J that was marked as an outlier using the original attribute rule set (org.apache.xml.utils.XMLStringDefault)

```
public void getChars(int srcBegin, int srcEnd, char dst[],
                     int dstBegin) {
  int destIndex = dstBegin;

  for (int i = srcBegin; i < srcEnd; i++) {
    dst[destIndex++] = m_str.charAt(i);
  }
}
```

Other cases of methods that are marked as outliers are not that clear. Listing 4.4

shows a method in ArgoUML that was identified as an outlier using the original attribute rule set. It is marked as an outlier because it violates the `inappropriate if included: Field Writer` rule. On examination of the method we see that it checks whether the value that is going to be returned is already created. If it not created it gets constructed before the value is returned. This is a known design pattern called *Lazy Initializer* [10] and, on inspection, is often used in ArgoUML. We would therefore consider this method as correctly named making this an incorrectly identified naming anomaly.

Listing 4.4: Listing of a method from ArgoUML that was incorrectly marked as an outlier using the original attribute rule set (org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram)

```
protected Action getActionComposition() {
  if (actionComposition == null) {
    actionComposition = makeCreateAssociationAction(
      Model.getAggregationKind().getComposite(),
      false, "button.new-composition");
  }

  return actionComposition;
}
```

4.4 Analysis

To find an answer to our question we have described the different steps in the process of finding method naming anomalies. Somewhat remarkable was the outcome of applying the threshold. By doing this we lost about a quarter of the methods in our corpus. Investigation of the lost methods showed that these methods contained the majority of the unique action-tokens in the corpus (see Table 4.3). Considering the task we were set out to do —finding methods that have bad names— it can be argued that specifically these methods might be methods that are not correctly named. At least, the meaning of these method names are not commonly known by the development community.

In other samples of identified outliers we see that whether or not the chosen action-token is correct, is often debatable. When we read code and investigate methods the decision whether a chosen name is appropriate is not a simple yes or no question. The appropriateness often lies somewhere in the middle between highly appropriate and completely inappropriate. Where we draw the line between good and bad depends on the eye of the beholder. What we did find is that the applied method does find methods with debatable names and more often also points out methods of which we would judge that they are badly named (see Figure 4.6). Although it still is hard to reason about the number of correctly identified methods with bad names, in this case we found a false positive rate of 50%, our result do indicate that using this method we will quicker identify methods with bad names compared to randomly selecting methods from a corpus.

Comparison between the original attribute rule set and the narrowed attribute rule

set show that the number of identified outliers go down drastically. But when we judge a sample from both sets we do not see a difference in the number of correctly identified methods bearing bad names. The biggest problem in this comparison is that we can not reason about the number of false negatives still remaining in our corpus. These are the methods the that have a bad name but were not identified as such. If we would like to be able to reason about this number we must have an application which contains methods of which the names are generally accepted as correct. To our knowledge such a 'golden standard' does not exists.

4.5 Conclusion

The question we were set out to answer in this chapter was:

Question 2. *Do methods that get identified by the method of Høst and Østvold have bad names?*

Yes, the method does find methods with bad names. It is imprecise by nature, however, and it is hard to find a measure for its imprecision. What we can say is that by using the described method the chance of quickly identifying methods with bad names is higher then by randomly selecting methods from a corpus. What the influence of the values of the attribute rule set is, is undecided. Less methods get identified but the number of true positives stays the same.

Chapter 5

An alternate approach

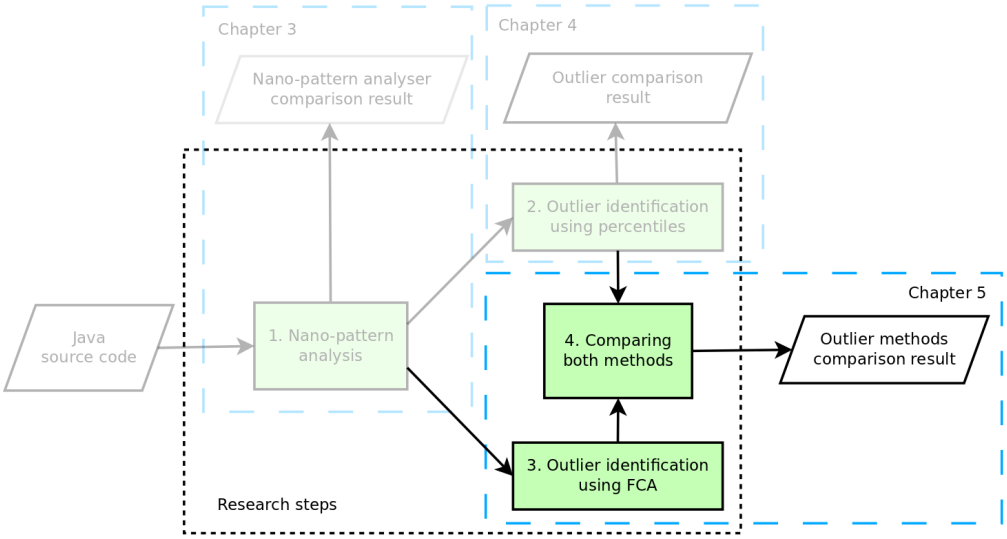


Figure 5.1: Step 3: Finding method naming anomalies using Formal Concept Analysis

In this chapter we introduce a new method of finding method naming anomalies using *Formal Concept Analysis* (FCA).

5.1 Rationale

In our previous chapter we have reproduced the approach of Høst and Østvold to identify method naming anomalies. We were able to identify method naming anomalies using the frequencies of nano patterns exhibited by action-token sets. These action-token sets contain all the methods that start with the same token. For instance, the action-token set of `get` contains all the methods that start with the verb `get`.

With the method of Høst and Østvold there are two possible reasons why a method is marked as an outlier:

- a method contains a pattern that is not often used by methods in the same action-token set
- a method does not contain a pattern that is often used by methods in the same action-token set

The method focusses on the presence or absence of separate patterns. What we do not see is the influence of combinations of patterns in methods.

Listing 5.1 shows an example of a method with a rare combination of nano patterns for a method beginning with the token `get`. It exhibits the patterns `Straight Line`, `Local Reader`, `Object Creator`, `Field Reader`, `Type Manipulator` and `Same Name`. According to the original attribute rule set (see Section 4.2.5) a method is considered in violation when it exhibits a pattern of which the occurrence in the corresponding action-token set is below 5%. If we take a look at the frequency set for `get` (see Figure C.1 in Appendix C) we see that the frequencies of the individual patterns that this method exhibits are all above the threshold of 5%. This means that this method is not in violation according to the method of Høst and Østvold.

Still, it is clear that the name this method has does not support its operation. Instead of returning an existing value it always creates a new object. We judge that the name `create` would therefore be better suited. What makes this method special is not that it includes or omits a certain nano pattern but the *combination* of patterns.[1]

Listing 5.1: Example of method with a rare combination of nano patterns (`org.tranql.ejb.ManyToManyCMR` in TranQL)

```
public Object get(InTxCache cache, CacheRow row) {
  return new ManyToManyRelationSet(cache, row.getId(), (Set) next.get(
      cache, row));
}
```

To be able to get insight in the interdependencies of the patterns we will use an alternate approach to identify naming anomalies. A way to find this rare combinations of the patterns is by using FCA. We will present a new method of identifying naming anomalies using FCA.

In the remainder of this chapter we will seek an answer to the following question:

Question 4. *How does our FCA method compare to the method of Høst and Østvold?*

5.2 Research method

We apply the steps as shown in Figure 5.2 to the action-token sets we constructed for the method described in the previous chapter. The outcome is that we will get another list of

[1] In this case the combination of the patterns `Straight Line`, `Local Reader`, `Object Creator`, `Type Manipulator` and `Same Name` only has an occurrence of 0.07% in the action-token set `get`

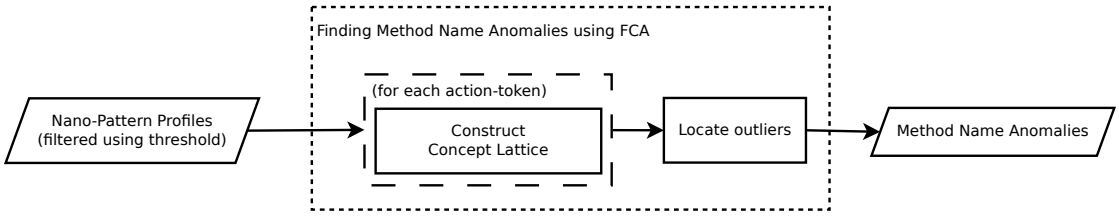


Figure 5.2: Overview of the different steps to identify method name anomalies using our FCA method

outliers. We can draw our conclusions by judging the identified outliers and comparing them with earlier results we found using the method of Høst and Østvold.

5.2.1 Formal Concept Analysis

In the coming section we will explain FCA. Our approach will be twofold. We will start with the formal definition followed by a more intuitive definition.

Formal definition

Formal Concept Analysis is a mathematical theory based on conceptual hierarchy [7]. It comprises of a set of well known techniques for the analysis of data. A *Formal Context* —a model at the heart of FCA— consists of objects $\mathcal{O}$ and attributes $\mathcal{A}$ and the relations between them $\mathcal{I}$. The relation between an object and an attribute can be written as $o\mathcal{I}a$ for $o \in \mathcal{O}$ and $a \in \mathcal{A}$. The set of all attributes that have a relations with a set $O \in \mathcal{O}$ is defined by $O' = \{a \in \mathcal{A} : o\mathcal{I}a \ \forall o \in O\}$. All objects that have a relations with the set $A \in \mathcal{A}$ is defined by $A' = \{o \in \mathcal{O} : o\mathcal{I}a \ \forall a \in A\}$.

A *Formal Concept* is the combination of (O, A) for which holds that $O' = A$ and $A' = O$. O is called the *extent* and A is called the *intent* of the concept $C = (O, A)$.

An ordering can be applied to concepts [6]. Let (O_1, A_1) and (O_2, A_2) be concepts of the context $\mathcal{C}(\mathcal{O}, \mathcal{A}, \mathcal{I})$, then (O_1, A_1) is a subconcept of (O_2, A_2) if $O_1 \subseteq O_2$ (which is equivalent to $A_1 \supseteq A_2$). Under the same conditions we can also say that (O_2, A_2) is a superconcept of (O_1, A_2). The relation $\leq$ is called the *hierarchical order* of the concepts [6]. This ordered set of concepts $\mathcal{C}(\mathcal{O}, \mathcal{A}, \mathcal{I}; \leq)$ is called a *Concept Lattice*.

Intuitive definition

FCA gives insight in the relations between attributes and objects. Table 5.1 shows a matrix of animals and some possible properties that might apply to them. If a cell contains an `x` it means that a certain property applies to the animal of that particular row. Although it is possible to see the shared properties amongst the animals it is hard to see all the underlying connections. By generating a concept lattice these combinations of attributes emerge.

Figure 5.3 shows the concept lattice associated with Table 5.1. The lattice shows all the explicit and latent relationships that exists between the animals and their properties. Each node in the lattice corresponds with a concept. In this lattice a concept consists out of a list of animals (the objects) and their accompanying properties (the attributes) they share. For instance, concept A contains the animal `Monkey` as its objects and the property `Has hands` as its attribute.

The lattice is hierarchically ordered. This means that the hierarchy of the concepts in the lattice matter. All the attributes that are applicable to the superconcepts reachable from a certain concept apply to this concept (the intent of a concept). An example can explain this.

The properties that apply to concept B in our lattice (see Figure 5.3) are `Can fly`, `Has wings` and `Has beak`. Although concept B does not have direct properties related to it, traversing through its superconcept relations we find all the properties that apply to the animal of the concept; the `eagle`. An examination of the context (Table 5.1) shows that this is indeed the case; the `eagle` has the properties `Can fly`, `Has beak` and `Has wings`.

To get the extent of a concept we have to traverse the subconcept relation. By doing this we find all the objects which also are applicable for a certain concept.

Concept C in the lattice (Figure 5.3) does not contain any direct objects. It does however contain the attribute `Has wings`. If we traverse the subconcept relation we find concepts which contain the animals `Penguin`, `Bat` and `Eagle`. An examination of the context (Table 5.1) shows that these are the only animals that share the property `Has wings`.

Animal \ Property	Breathes in water	Can fly	Has beak	Has hands	Has wings
Bat		x			x
Eagle		x	x		x
Monkey				x	
Parrot Fish	x		x		
Penguin			x		x

Table 5.1: A simple context containing animals and their attributes

Why use FCA?

We hypothesise that a group of methods starting with the same action-token should, next to being consistent in the the separate nano patterns it exhibits, also be consistent in the combination of nano patterns that are used. Finding methods that exhibit rare combinations of nano patterns can therefore also be seen as methods that break consistency. Methods that have rare combinations of nano patterns are methods that do different operations than other methods with similar names. FCA allows us to explore the explicit and latent relations that exists between a group of methods and the nano patterns they exhibit.

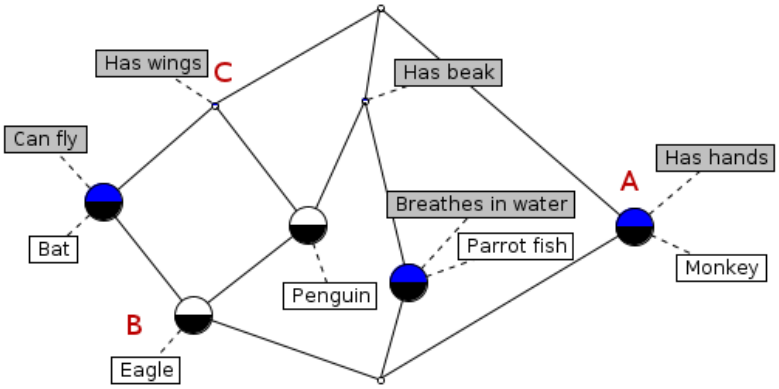


Figure 5.3: A FCA lattice constructed from the context shown in Table 5.1 visualized via ConExp (http://conexp.sourceforge.net)

5.2.2 Constructing Formal Contexts

We will construct a formal context for each action-token set. The individual method names of the methods in an action-token set will be used as the object set of a context. The attribute set of the context consists out of the different nano patterns. A formal concept therefore consist of a combination of methods starting with the given action-token (the objects) and the nano patterns they share (the attributes). Table 5.2 shows an excerpt of a context constructed using the method names of an action-token set and the nano patterns they posses.[2]

The action-token sets which we use to construct formal contexts are the same sets that we used for the analysis with the method of Høst and Østvold. These are the action-token sets that we are left with *after* we applied the threshold as described in section 4.2.4.

Nano pattern / Method	No Return	Type Manipulator	...	Straight Line
doTypeCheck()		x	...	x
doCheckOfObject(Object)	x		...	x
doCheckAndCast(Object)	x	x	...	
doCast()	x	x	...	x

Table 5.2: An excerpt of a formal context constructed for methods that start with the action-token `do`

[2]The methods used to populate the formal context shown in Table 5.2 are all taken from our test project and are not part of our corpus.

5.2.3 Building Concept Lattices

We construct concept lattices out of the formal contexts. Figure 5.4 shows an example of such a concept lattice. By constructing this lattice we may see latent relations between the nano patterns and the methods of an action-token set. For instance, by examining the lattice we see that there is a concept (denoted with the letter A in the Figure 5.4) consisting of the combination of the nano patterns `Straight Line`, `Leaf`, `No Param` and `No Return` with the methods `doCast()` and `doCheckOfObject(Object)`. There is no direct method or nano pattern associated with this concept but it still exists.

Seeing these kind of relations is harder when we do not have access to the concept lattice. What the lattice shows are all the unique relations that exists between a given set of objects and its attributes, or, in our case, a set of methods and the nano patterns they contain. We use this property of the concept lattice to locate our outliers.

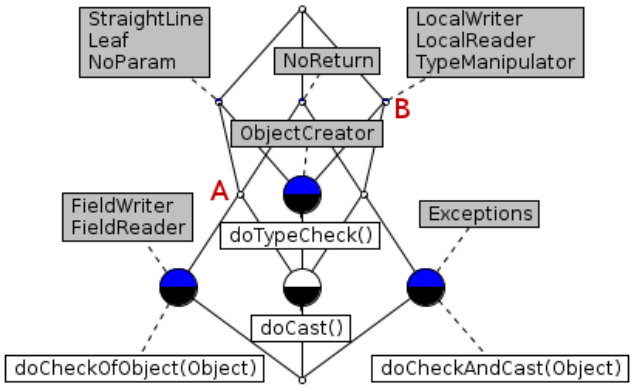


Figure 5.4: Example of a concept lattice for methods starting with the action-token `do`. It shows the method names (the objects of a concept) and the associated nano patterns (the attributes of a concept). Table 5.2 shows a part of the formal context

5.2.4 Finding outliers using FCA

In our constructed concept lattices we want to find concepts that contain rare combinations of nano patterns compared to the other methods in the action-token set. By focussing on concepts that have a low amount of methods we can locate methods which have these rare combinations of nano patterns.

We calculate the percentage of the objects of each concept in regards to the total number of objects in the context. For instance, concept A in Figure 5.4 contains the objects `doCast` and `doCheckOfObject(Object)`. Considering that the context is constructed with 4 methods means that concept A contains 50% of all the methods in the context. Con-

cept B holds 3 methods namely `doTypeCheck()`, `doCast` and `doCheckAndCast(Object)` meaning that this concept holds 75% of all the methods in the context.

We define a percentage threshold, a minimum number of methods that a concept should contain before it is considered an outlier. The example above does not contain many concepts but the number of concept quickly grow depending on the number of methods and present nano patterns. Since our biggest action-token set, the action-token set for `get`, contains 37847 methods we expect that we can have a high number of concepts. Given the above we define, after some initial experimentation, that a concept should at least contain **0.15%** of all the methods in the context. The methods contained by a concept are considered in violation when the concept contains less then 0.15% of all the methods of the context.

5.2.5 Qualifying the result

Comparing both methods

To validate the results we will compare the outcome of our FCA method with the outcome of the method of Høst and Østvold. This gives an overview of the statistics on the found outliers for both methods. We will compare the number of found outliers for both methods. Next to this we will examine the overlap in found outliers between both methods. This gives us insight in the number of outliers that were found with our FCA method but were not located using the method of Høst and Østvold or vice versa.

Qualifying the found outliers

Like we did for the result found by the method of Høst and Østvold we will judge a random sample taken from the outlier list found by our FCA method mixed with random samples taken from the list that were not marked as outliers. Again, when we judge the samples it is unknown to us which of the samples were marked as outliers and which not. For the random samples from the FCA method we will only use those methods that are not marked as outliers by the method of Høst and Østvold but were marked as such by our FCA method. By doing this we can measure the impact of our FCA method separately from the method of Høst and Østvold.

5.3 Results

5.3.1 Comparing both methods

Table 5.3 shows the result of comparing our FCA method with the method of Høst and Østvold. Out of the total methods, 2.67% get marked as outliers by our FCA method. This is just under the number of outliers that get found when using the narrowed attribute rule set.

If we do a outlier comparison on the outliers that are found by both methods we see that compared to the method using the original attribute rule set almost 70% of

	FCA		Original attribute rule set			Narrowed attribute rule set		
Action	# methods	% of total	# methods	% of total	# shared	# methods	% of total	# shared
get	1607	4.25	3969	10.49	1165	1306	3.45	620
set	527	3.20	2398	14.56	420	647	3.93	278
is	181	2.89	544	8.68	151	117	1.87	65
add	165	4.02	433	10.55	94	39	0.95	20
create	158	3.88	183	4.49	53	15	0.37	7
to	86	2.83	201	6.62	55	86	2.83	20
visit	32	1.66	133	6.91	16	49	2.55	10
remove	39	2.04	67	3.50	16	67	3.50	16
has	28	1.85	194	12.81	28	96	6.34	25
read	23	2.05	14	1.25	6	14	1.25	6
Total	**2922**	**2.67**	**9991**	**9.12**	**2044**	**3160**	**2.88**	**1087**

Note: The '# shared' column relates to the number of shared methods that are found by both the FCA method and the method of Høst and Østvold

Table 5.3: Comparison of FCA method with the method of Høst and Østvold (both original as narrowed attribute rule set) for the top ten action-tokens

the outliers get found by both methods. Compared to the method using the narrowed attribute rule set this is less, they share a little more then 37% of their outliers.

The action-token that has the most shared outliers is `has`. All the outliers found using FCA are also marked as an outlier using the original attribute rule set and in 89% of the cases using the narrowed attribute rule set.

If we take a look at the number of outliers that get found per action-token set we see that the amount of found outliers, absolute and relative, are less for smaller action-token sets. Further investigation shows that if the action-token set contains less than a 1000 methods no more outliers get reported.

5.3.2 Validating the found outliers

Figure 5.5 shows the outcome of the random sample test with our FCA method. Out of the 30 samples that were marked as outliers with our FCA method we judged that 12 methods had a bad name. The other 18 cases we deemed correctly named.

The non-outlier list shows similar results as earlier although we slightly judged more methods on the non-outlier list to be incorrectly named as before. This time we correctly judged that 23 out of the 30 cases. The other 7 cases had, according to our judgement, bad names. Table D.2 in Appendix D contains the samples and our reasoning.

5.4 Analysis

5.4.1 Comparing the methods

Number wise we see that our FCA method find a similar amount of outliers compared to the method of Høst and Østvold using the narrowed attribute rule set. Some of the

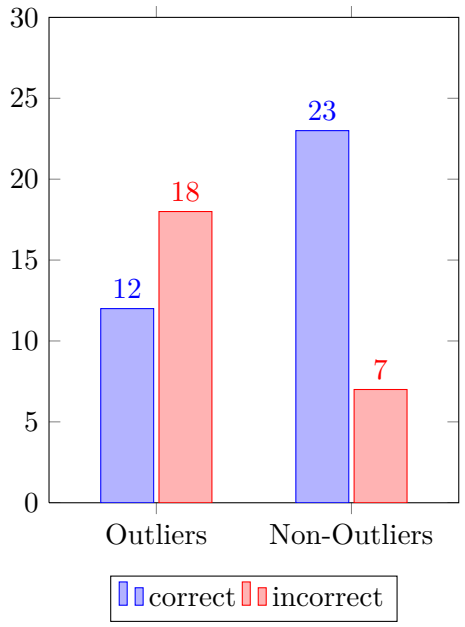


Figure 5.5: Correctly and incorrectly identified naming outliers using the FCA method

identified methods using our FCA method also get identified with the method of Høst and Østvold. Next to this overlap there are methods that only get identified using our FCA method. If we judge these outliers we do find that some are indeed methods that have bad names.

Our judgement experiment showed that the rate of correctly identified naming anomalies using our FCA method was lower than the earlier found results judging the outliers that were found using the method of Høst and Østvold. Whether or not we would see the same results if we would repeat the test is hard to say. We would have to expand our experiment before we can draw any conclusions on this.

5.4.2 Examining a found outlier

Listing 5.2 shows an outlier which gets found using our FCA method but not using the method of Høst and Østvold. This method reads objects from an input stream and adds them to an inner map. Although the method bears the name `read` this only partially describes what the method does. After the object is read it is added to the map. Another noteworthy detail about this method is that is actually does not read a *single* object although the name of the method does let us to belief. The method continues on reading objects from the input stream as long as they are available. These two arguments let us to believe that this method has a bad name.

The method is not found using the method of Høst and Østvold. An inspection of the frequency set of the action-token `read` explains why this is the case (see Figure C.10 of Appendix C). The frequencies of the nano patterns in the action-token set are none

distinctive. Almost all frequencies lie between 5% and 95%. Only the occurrence of the pattern `Recursive` is distinctive with a frequency of 1.25%.

That these kind of methods gets identified using our FCA method supports our intuition that the combination of rare nano patterns also is an indicator of methods that have bad names.

Listing 5.2: An example of a found outlier using our FCA method that was not marked as an outlier using the original attribute set method of Høst and Østvold (`org.apache.commons.collections.ReferenceMap` of Commons Collections)

```
private void readObject(ObjectInputStream inp) throws IOException,
    ClassNotFoundException {
  inp.defaultReadObject();

  table = new Entry[inp.readInt()];
  threshold = (int)(table.length * loadFactor);
  queue = new ReferenceQueue();
  Object key = inp.readObject();

  while (key != null) {
    Object value = inp.readObject();
    put(key, value);
    key = inp.readObject();
  }
}
```

5.4.3 Limitations of the used method

The fact that no more outliers get reported when the number of methods in an action-token set fall under a certain limit is probably due to the static nature of our applied occurrences threshold. Our method reports an outlier when a concept contains less than 0.15% of the methods in the complete action-token set. Depending on the absolute amount of methods and the stability of the occurring nano patterns of these methods it could be that the lattice simply does not contain concepts that break this rule. A simple example helps to explain our point.

The minimal amount of methods (objects) that is contained by a concept is 1. Now let us build an lattice out of an action-token set that consists out of 100 methods. In this lattice the concepts with the least contained methods (1) still contain 1% of the total amount of methods in the action-token set. Therefore there will never be any outliers found by our FCA method for these smaller action-token sets. A possible way around this problem is to make the applied threshold dynamic. Instead of applying the same threshold for every size of action-token set we could determine the height of the threshold based on the amount of methods in the set.

5.5 Conclusion

In this chapter we wanted to answer the following question:

Question 4. *How does our FCA method compare to the method of Høst and Østvold?*

Our FCA method finds outliers that are not recognized using the method of Høst and Østvold. Similar to the results we saw for the method of Høst and Østvold these outliers also have a higher change of being methods that have bad names. This indicates that rare combinations of nano patterns are also an indicator for methods that have bad names.

Chapter 6

Discussion

We were set out to acquire a deeper understanding of method naming anomalies and how they could be located.

In our research we have investigated the relation between the occurrence of nano patterns and the used action-token of the method name. We have retraced and reproduced earlier work done by Høst and Østvold on finding inconsistencies of present nano patterns in methods and their name. Next to this method we have introduced an alternate approach of identifying method name anomalies using FCA. With this method we were able to investigate whether rare combinations of nano patterns are also a pointers to methods that have bad names. The outcome was that both methods do seem to find methods with debatable names.

The experiments that we have done indicate that method naming anomalies have a higher change of being methods having bad names than compared to random selected methods from a corpus. Whether this truly scales to the whole of the (JAVA) development community is hard to say. If we want to able to prove this we should repeat our outlier judging experiment with many more developers. With our current research we can say that we have indications that a method name anomaly hints that there is a mismatch between the chosen name and the method implementation.

6.1 Threats to validity

6.1.1 The diversity of the corpus

A possible threat is that our corpus was skewed onto a certain application domain or development community. To prevent this we have chosen to use applications taken from different application domains and from different sources. All applications were part of earlier research on nano patterns or finding method naming inconsistencies

6.1.2 Experimenter bias

As part of our experiments we have reviewed outlier samples and judged them on their correctness. To prevent experimenter bias we have mixed the random samples taken from the outlier list with random samples from the non-outlier list. These lists were all constructed automatically without our intervention. While we reviewed the samples it was unknown to us if a sample was marked as an outlier or not.

6.2 Future work

6.2.1 Finding Naming Anomalies

Influence of the nano pattern catalogue

We have chosen to use the nano pattern catalogue as introduced by Singer et al.[26]. Høst and Østvold used different catalogues of nano patterns in the course of their research[13, 14, 15, 16]. Seeing that we have a clear difference in found outliers with the method introduced by [14] it is interesting to investigate what the influence of the different nano pattern catalogues is.

Influence of the corpus

In our research we have focused on a corpus that contained applications from different domains. It is interesting to see whether we see the same results using a corpus containing only code from one company or one development community. It could be that certain development and naming styles are detectable is we only focus on these applications.

6.2.2 Handling Naming Anomalies

One of the questions we have not touched in our research is what we should *do* with the found method naming anomalies. How should they be handled? Simply changing the name of the method is not always the solution to the found problem. Sometimes the invalidity of the name is an indication for more structural problems. In those cases it could be more advisable to change the implementation of the method then just the name. These cases could also be considered as more general *code smells*. We came across some of these cases in our search (see Listing 6.1).

Some research had been done by Høst and Østvold into this problem[14]. In their research into *Naming Bugs* they came up with a method of suggesting name improvements based on the method phrase frequency sets they constructed. This could be a suitable solution for the first part of the problem -a method that has a wrong name- but this does not fix the second part of the problem were the method implementation itself is incorrect.

Listing 6.1: An example of a found outlier where the underlying problem is maybe not the name but the whole structure chosen to solve a certain problem. This method was marked as outlier by the FCA method because it has a rare combination of the `Leaf`, `Object Creator`, `Field Reader`, `No Param` and `Type Manipulator` nano patterns for methods that start with `get`

```
public Exception getLinkedException() {
  // jason: this is bad, but whatever... the jms folks
  // should have had more insight

  if (nested == null)
    return this;

  if (nested instanceof Exception) {
    return (Exception) nested;
  }

  return new NestedException(nested);
}
```

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in large databases. In *Proceedings of the International Conference on Management of Data*, pages 207–216. ACM SIGMOD Record, 1993.

[2] E. Bruneton. ASM 4.0 - A Java bytecode engineering library. Technical report, OW2 Consortium, 2011.

[3] R. Buse and W. Weimer. A metric for software readability. *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, page 121, 2008.

[4] S. Butler, M. Wermelinger, and Y. Yu. Improving the tokenisation of identifier names. *ECOOP 2011-Object-Oriented Programming*, pages 130–154, 2011.

[5] C. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 112–122. IEEE, 1999.

[6] C. Carpineto and G. Romano. *Concept Data Analysis: Theory and Application*. Wiley, 2004.

[7] S. Colton and D. Wagner. Using Formal Concept Analysis in Mathematical Discovery. *Towards Mechanized Mathematical Assistants*, pages 205–220, 2007.

[8] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.

[9] F. Deissenboeck and D. Ratiu. A unified meta-model for concept-based reverse engineering. *Proc. 3rd International Workshop on Metamodels*, 2006.

[10] M. Fowler. *Patterns of Enterprise Application Architecture*. Number 2 in The Addison-Wesley signature series. Addison-Wesley Professional, 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison Wesley, 1995.

[12] J. Gil and I. Maman. Micro patterns in Java code. *ACM SIGPLAN Notices*, 40(10): 97, October 2005.

[13] E. Høst and B. Østvold. The Programmer's Lexicon, Volume I: The Verbs. *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 193–202, September 2007.

[14] E. Høst and B. Østvold. Debugging method names. *ECOOP 2009-Object-Oriented Programming*, pages 294–317, 2009.

[15] E. Høst and B. Østvold. The Java programmer's phrase book. *Software Language Engineering*, pages 322–341, 2009.

[16] E. Høst and B. Østvold. Canonical Method Names for Java. *Proceedings of the Third international conference on Software language engineering*, pages 1–20, 2011.

[17] D. Lawrie, H. Feild, and D. Binkley. Syntactic Identifier Conciseness and Consistency. *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 139–148, September 2006.

[18] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12, 2006.

[19] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*. ACM, 2006.

[20] D. Low. Protecting Java code via code obfuscation. *Crossroads*, 4(3):21–23, 1998.

[21] S. McConnell. *Code Complete*. Microsoft Press, second edition, 2004.

[22] Sun Microsystems. Java Code Conventions. Technical report, Sun Microsystems, 1997.

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

[24] S. Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1):143–192, 2000.

[25] J. Singer and C. Kirkham. Exploiting the Correspondence between Micro Patterns and Class Names. *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 67–76, September 2008.

[26] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis. Fundamental Nano-Patterns to Characterize and Classify Java Methods. *Electronic Notes in Theoretical Computer Science*, 253(7):191–204, September 2010.

[27] A. Thies and C. Roth. Recommending rename refactorings. *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering - RSSE '10*, pages 1–5, 2010.

[28] G.M. Weinberg. *The Psychology of Computer Programming.* Computer Science Series. Van Nostrand Reinhold, 1971.

[29] L. Wittgenstein. *Philosophical Investigations*, volume 34. Blackwell, 1953.

Appendices

Appendix A

Overview of Java AST nodes in Rascal

The following AST nodes are defined in Rascal. These nodes can be present in the Java AST and function as a base for our nano pattern source code analyser.

Table A.1: Defined Java AST nodes in Rascal

Type	Node	Properties
Declarations	Anonymous Class Declaration	list of Nodes: body declarations
	Annotation Type Declaration	list of Modifiers: modifiers, string: name, list of Nodes: body declarations
	Annotation Type Member Declaration	list of Modifiers: modifiers, Node: type argument, string: name, list of Nodes: default block*
	Enum Declaration	list of Modifiers: modifiers, string: name, list of Nodes: implements, list of Nodes: enum constants, list of Nodes: body declarations
	Enum Constant Declaration	list of Modifiers: modifiers, string: name, list of Nodes: arguments, list of Nodes: Anonymous Class Declaration*
	Type Declaration	list of Modifiers: modifiers, string: object type, string: name, list of Nodes: generic types, list of Nodes: extends*, list of Nodes: implements, list of Nodes: body declarations
	Field Declaration	list of Modifiers: modifiers, Node: type, list of Nodes: fragments
	Initializer	list of Modifiers: modifiers, Node: body
	Method Declaration	list of Modifiers: modifiers, list of Nodes: generic types, list of Nodes: return type*, string: name, list of Nodes: parameters, list of Nodes: possible exceptions, list of Nodes: implementation*

(* indicates an optional field)

Continued on next page

Table A.1 Defined JAVA AST nodes in RASCAL – continued from previous page

Type	Node	Possible properties
	Import Declaration	string: name, boolean: static import, boolean: on demand
	Package Declaration	string: name, list of Nodes: annotations
	Single Variable Declaration	string: name, list of Modifiers: modifiers, Node: type, list of Nodes: initializer*, boolean: is varargs
	Variable Declaration Fragment	string: name, list of Nodes: initializer*
	Type Parameter	string: name, list of Nodes: extendsList
Expressions	Marker Annotation	string: type name
	Normal Annotation	string: type name, list of Nodes: member value pairs
	Member Value Pair	string: name, Node: value
	Single Member Annotation	string: type name, Node: value
	Array Access	Node: array, Node: index
	Array Creation	Node: type, list of Nodes: dimensions, Node initializer*
	Array Initializer	list of Nodes: expressions
	Assignment	Node: left side, Node: right side
	Boolean Literal:	boolean: value
	Cast Expression	Node: type, Node: expression
	Character Literal	string: value
	Class Instance Creation	Node expression*, Node: type, list of Nodes: generic types, list of Nodes: typed arguments, Node anonymous class declaration*
	Conditional Expression	Node: expression, Node: then branch, Node: else branch
	Field Access	Node: expression, string: name
	Infix Expression	string: operator, Node: left side, Node: right side, list of Nodes: extended operands
	Instanceof Expression	Node: left side, Node: right side
	Method Invocation	Node expression*, list of Nodes: generic types, string: name, list of Nodes: typed arguments
	Super Method Invocation	string qualifier*, list of Nodes: generic types, string: name, list of Nodes: typed arguments
	Qualified Name	string: qualified name
	Simple Name	string: simple name
	Null Literal	
	Number Literal	string: number
	Parenthesized Expression	Node: expression
	Postfix Expression	Node: operand, string: operator
	Prefix Expression	Node: operand, string: operator
	String Literal	string: string value
	Super Field Access	string qualifier*, string: name
	This Expression	string qualifier*:
	Type Literal	Node: type

(* indicates an optional field)

Continued on next page

Table A.1 Defined JAVA AST nodes in RASCAL – continued from previous page

Type	Node	Possible properties
	Variable Declaration Expression	list of Modifiers: modifiers, Node: type, list of Nodes: fragments
Statements	Assert Statement	Node: expression, Node message*
	Block	list of Nodes: statements
	Break Statement	string label*
	Constructor Invocation	list of Nodes: generic types, list of Nodes: typed arguments
	Super Constructor Invocation	Node expression*, list of Nodes: generic types, list of Nodes: typed arguments
	Continue Statement	string label*
	Do Statement	Node: body, Node: while expression
	Empty Statement	
	Enhanced For Statement	Node: parameter, Node: collection expression, Node: body
	Expression Statement	Node: expression
	For Statement	list of Nodes: initializers, Node boolean expression*, list of Nodes: updaters, Node: body
	If Statement	Node: boolean expression, Node: then statement, Node else statement*
	Labelled Statement	string: label, Node: body
	Return Statement	Node expression*
	Switch Statement	Node: expression, list of Nodes: statements
	Switch Case	boolean: is default, Node expression*
	Synchronized Statement	Node: expression, Node: body
	Throw Statement	Node: expression
	Try Statement	Node: body, list of Nodes: catch clauses, Node finally
	Catch Clause	Node: exception, Node: body
	Type Declaration Statement	Node: Type Declaration
	Variable Declaration Statement	list of Modifiers: modifiers, Node: type, list of Nodes: fragments
	While Statement	Node: expression, Node: body
Types	Array Type	Node: type of array
	Parameterized Type	Node: type of param, list of Nodes: generic types
	Qualified Type	string: qualifier, string: name
	Primitive Type	string: primitive
	Simple Type	string: name
	Wildcard Type	Node wildcard type*, string bound*
Comments	Block Comment	
	Line Comment	
Javadoc	Javadoc	
	Tag Element	
	Text Element	
	Member Ref	
	Member Ref Parameter	

(* indicates an optional field)

Continued on next page

Table A.1 Defined JAVA AST nodes in RASCAL – continued from previous page

Type	Node	Possible properties

Appendix B

Definition of the Nano Patterns as implemented in the source code analyser

This appendix contains the definitions of the nano patterns as they are implemented in the RASCAL source code analyser. The names type setted in `monospace` are references to nodes or attributes of the nodes as described in appendix A

Table B.1: Definition of the Nano Patterns in the source code analyser

Nano Pattern	Definition
No Param	True if the `list of parameters` of the `method declaration` is empty
No Return	True if the `list of return type` of the `method declaration` is empty or void
Recursive	True if the `method body` contains a `method invocation` on the current object with the `same method name` and `parameter types` as the current method.
Same Name	True if the `method body` contains a `method invocation` on another object with the same name as the current method **or** if the `method body` contains a `method invocation` on the same object with the same name but with different `parameters` **or** if the `method body` contains a `super method invocation` with the same name as the current method.
Leaf	True if the `method body` does not contain a `method invocation`, `super method invocation`, `constructor invocation` or `super constructor invocation`.
Object Creator	True if the `method body` does not contain a `class instance creation`

Continued on next page

Table B.1 Definition of the Nano Patterns in the source code analyser

Nano Pattern	Definition
Field Reader	True if the `method body accesses a field` on the current or another object **or** if the right hand side of an `assignment` contains a reference to a field (a `simple name`) of the current object (which is not defined within the scope of the method)
Field Writer	True if the left hand side of an `assignment accesses a field` on the current or another object **or** the left hand side of an `assignment` contains a reference to a field (a `simple name`) on the current object (which is not defined within the scope of the method)
Type Manipulator	True if the `method body` contains a `cast` or an `instanceof` expression
Straight line	True if the `method body` does not contain a `for` or `switch` statement, a `conditional` expression or a `catch clause`
Looping	True if the `method body` contains a `for`, `enhanced for`, `while` or `do while` statement
Exceptions	True if the `list of possible exceptions` of the `method declaration` is not empty
Local Reader	True if the `method body` contains a reference to a field (a `simple name`) which is declared within the scope of the current method and is not part of the left hand side of an `assignment`
Local Writer	True if the `method body` contains a `assignment` of which the left hand side contains a reference to a field (a `simple name`) which is declared within the scope of the current method **or** the `method body` contains a `variable declaration`
Array Creator	True if the `method body` contains an `array creation`
Array Reader	True is the `method body` contains an `array access` or an `enhanced for statement` of which the `collection` it iterates over is of type array which are not part of left hand side of an assignment.
Array Writer	True if the `method body` contains an `array initializer` or the left hand side of an `assignment` contains an `array access`

Appendix C

Pattern frequencies for the top ten action-tokens

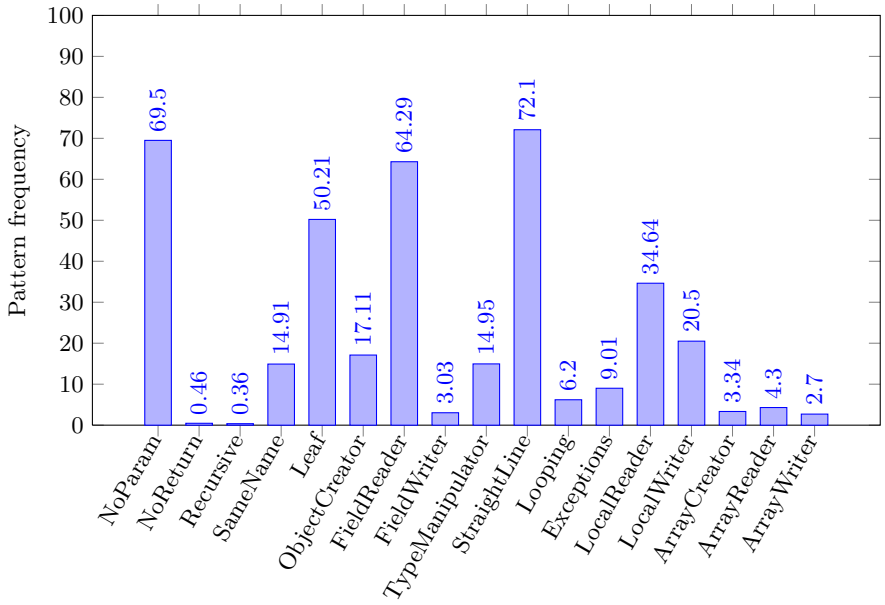


Figure C.1: Frequency of the different nano-patterns for methods that start with *get* (calculated over 37847 methods)

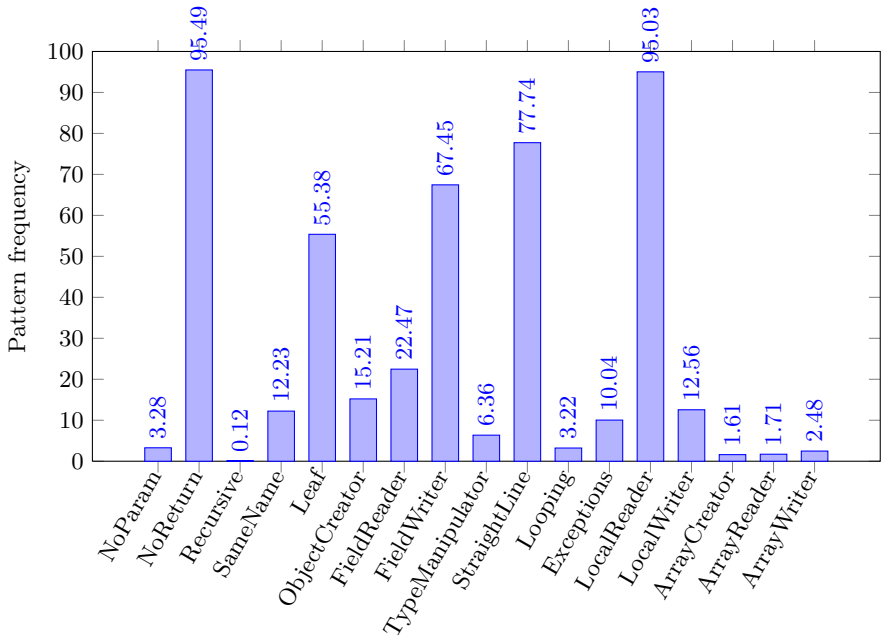


Figure C.2: Frequency of the different nano-patterns for methods that start with *set* (calculated over 16472 methods)

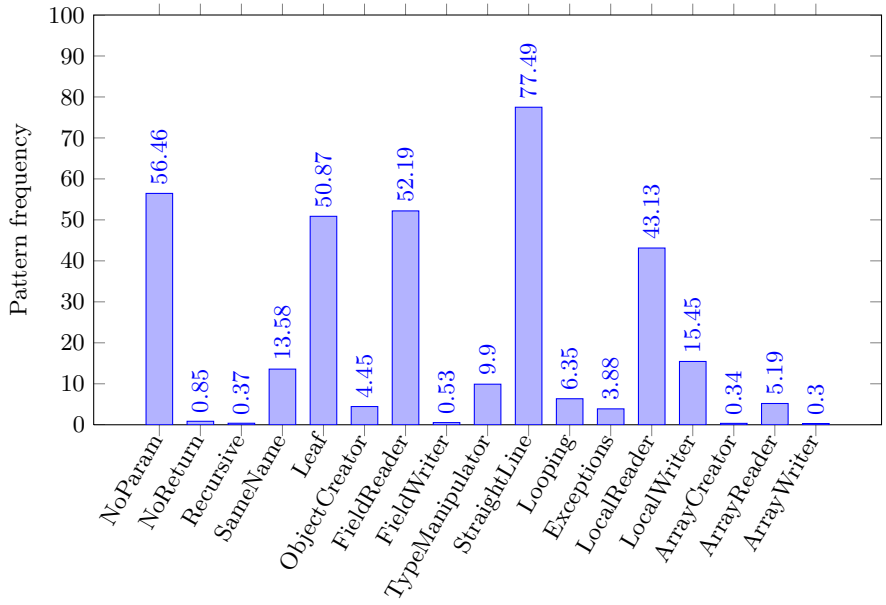


Figure C.3: Frequency of the different nano-patterns for methods that start with *is* (calculated over 6265 methods)

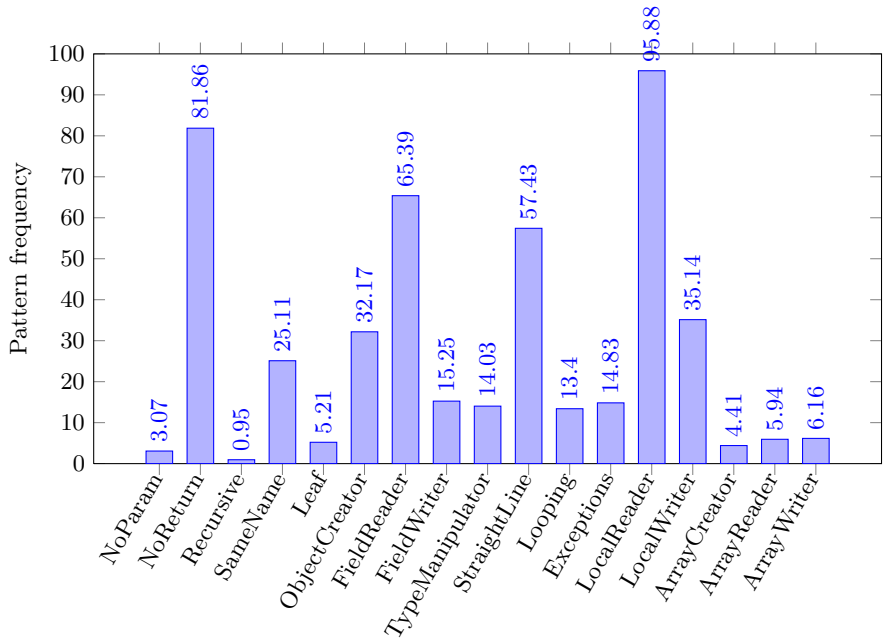


Figure C.4: Frequency of the different nano-patterns for methods that start with *add* (calculated over 4106 methods)

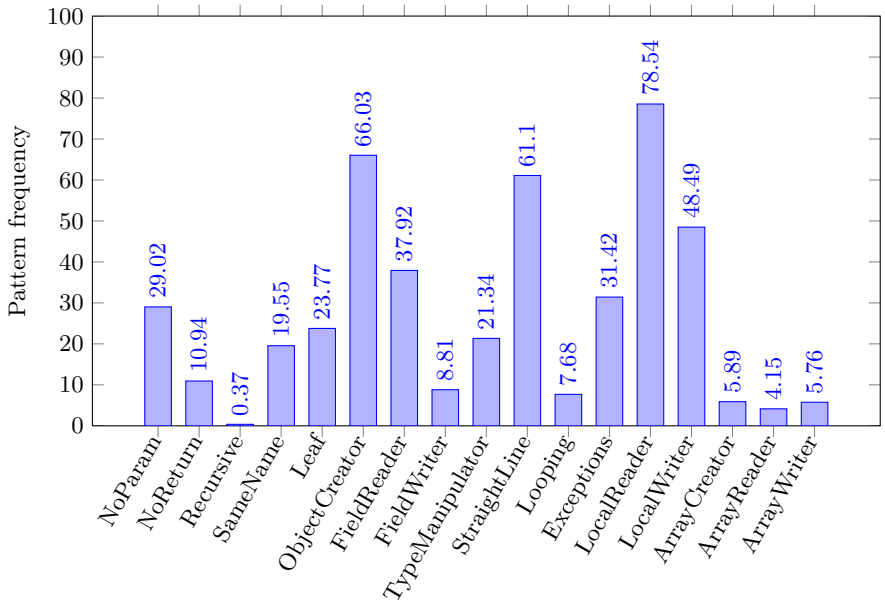


Figure C.5: Frequency of the different nano-patterns for methods that start with *create* (calculated over 4077 methods)

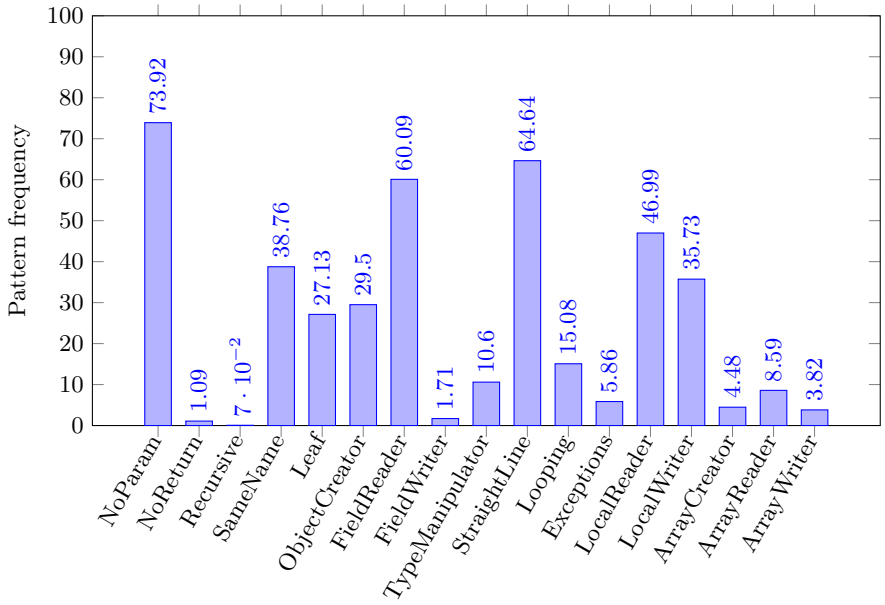


Figure C.6: Frequency of the different nano-patterns for methods that start with *to* (calculated over 3037 methods)

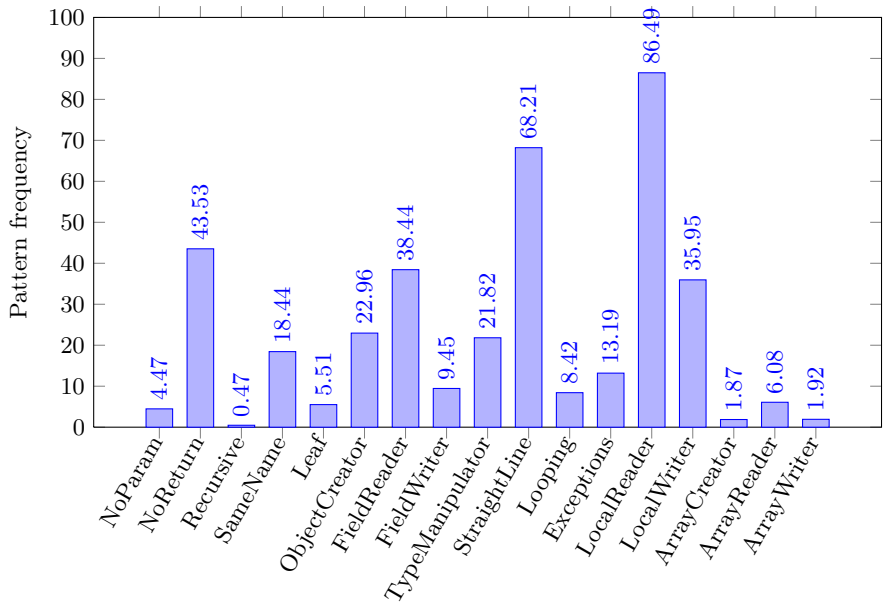


Figure C.7: Frequency of the different nano-patterns for methods that start with *visit* (calculated over 1925 methods)

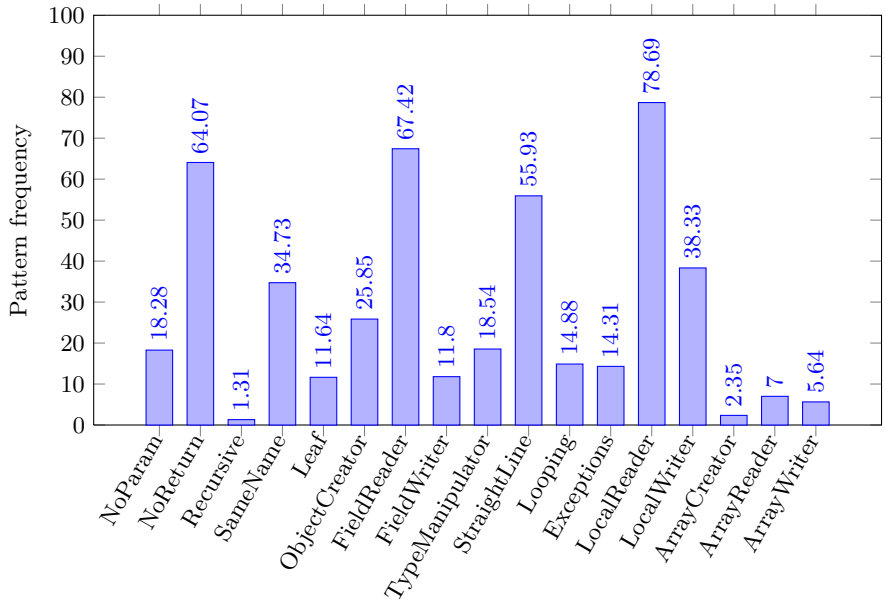


Figure C.8: Frequency of the different nano-patterns for methods that start with *remove* (calculated over 1915 methods)

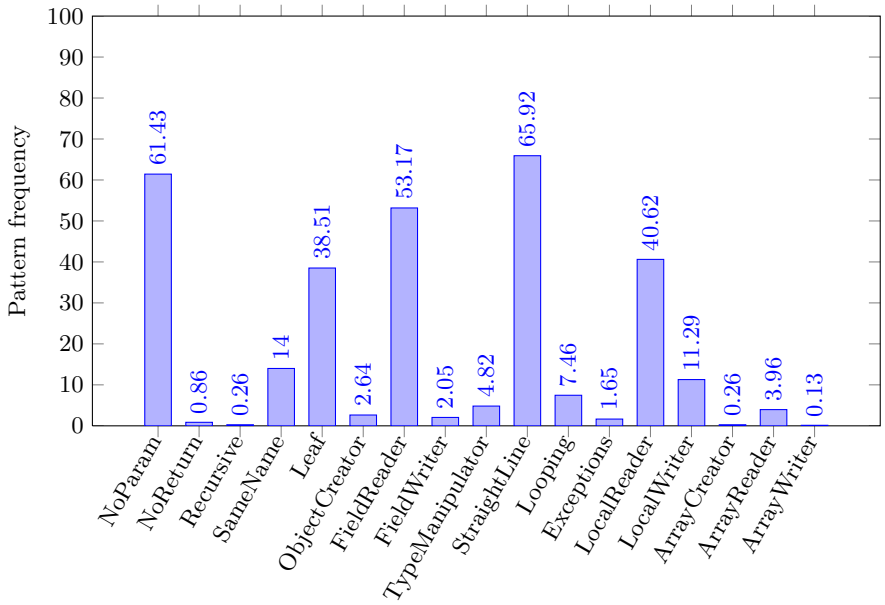


Figure C.9: Frequency of the different nano-patterns for methods that start with *has* (calculated over 1514 methods)

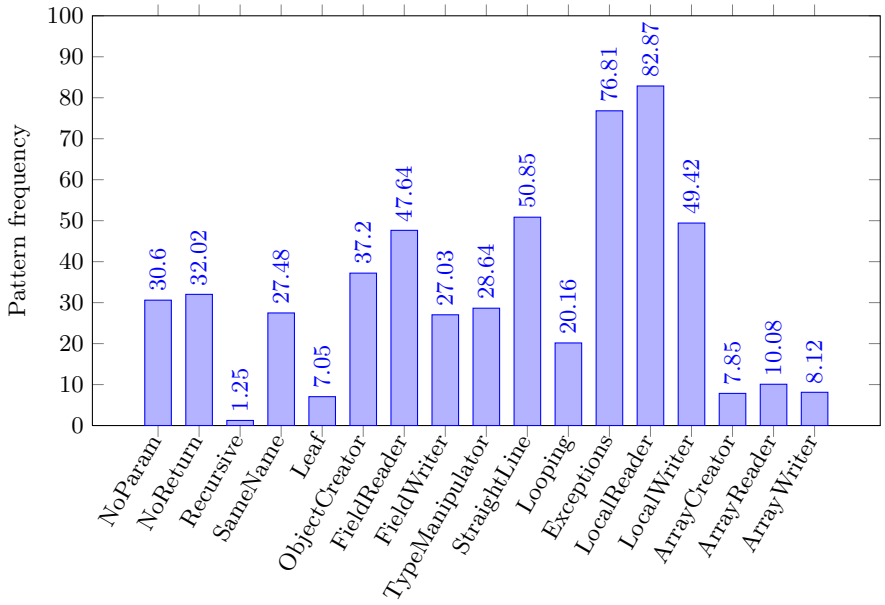


Figure C.10: Frequency of the different nano-patterns for methods that start with *read* (calculated over 1121 methods)

Appendix D

Review of the samples

This appendix contains the overviews of the judgement of the different samples that were taken from the outlier and non-outlier groups. There are three tables in total. The first contains the judgement of the random samples using the Høst and Østvold method with the original attribute rule set [14]. The second table contains the judgement of the random samples using the Høst and Østvold method with the narrowed attribute rule set (see 4.2.5). The third table contains the judgement of the random samples using our FCA method. Table D.1 describes the meaning of the different columns used.

Table D.1: Meaning of the different columns

Column name	**Explanation**
G	The group the sample belongs to. Can have the value *O* for *Outlier group* or *C* for *Control group*. It was unknown at the time of judgement in which group a sample belonged to.
Cor	Whether or not the judgement was correct according to the sample group. For instance: if a method name was judged invalid and the sample belonged to the outlier group it was marked as correct and vice versa. It was also marked correct if the method name was deemed valid and it belonged to the control group.
Project	The project the sample belongs to.
Class	The fully qualified name of the class. The packages have been abbreviated to their first letter.
Method	The name of the method including the types of its arguments.
Appr	Whether or not we judged the method name to be appropriate and in line with its implementation.
Cert	The certainty of our judgement. *1* meaning that we are certain. *2* meaning we are not that certain.
Reason	A brief description why we came to this conclusion. Sometimes just contains a description of the method implementation.

Table D.2: Result of the review of the *Høst and Østvold method* using the *orgininal attribute rule set*

G	Cor	Project	Class	Method	Appr	Cert	Reason
C	Yes	Antlr	a.DefineGrammarSymbols	getHeaderActionLine(String)	Yes	1	Method finds the line a so called header token is contained. If it is there it will return the line number
C	Yes	FitNesse	f.d.Loop	setupDecorator(array)	Yes	2	Does a setup using the input variables
O	Yes	FitNesse	f.s.HashWidgetConversionTestMapConstructor	setMap(Map(String,String))	No	2	This method is called set but returns a boolean. Furthermore, nothing gets actually set
O	No	FitNesse	f.w.VirtualEnabledPageCrawlerTest	setUp()	Yes	2	A normal test setup method
C	Yes	XML-Batik	o.a.b.a.s.JAuthenticator	run()	Yes	1	Method of anonymous inner class
O	Yes	XML-Batik	o.a.b.b.AbstractGraphicsNodeBridge	getBBox()	No	2	Lazy loaded get but does a lot of operations
O	No	XML-Batik	o.a.b.c.d.CSSOMSVGColor	getBlue()	Yes	1	Lazy loaded get
O	No	XML-Batik	o.a.b.c.p.Parser	parseRule()	Yes	2	A parser method that does dispatches to other parser methods
C	Yes	XML-Batik	o.a.b.d.s.SVGOMImageElement	newNode()	Yes	1	Create a new node, should use the verb
C	Yes	XML-Batik	o.a.b.d.s.SVGOMRect	getWidth()	Yes	2	Normal getter
O	Yes	XML-Batik	o.a.b.e.a.i.c.PNGImageEncoder	encode(RenderedImage)	No	1	This is a very large method (approx 200 lines). Should be spilt up in multiple methods that can be more precise
C	No	Com Col	o.a.c.c.CursorableSubList	listIterator()	No	2	The name itself is not a verb. In this case an iterator is created and returned
C	No	Com Col	o.a.c.c.FastArrayList	add(Object)	No	1	This add lets you insert a element on a certain location. Normally add will add it to the collection without specifying a location
C	No	Com Col	o.a.c.c.m.AbstractHashedMap	convertKey(Object)	No	2	This only converts the key if its null it then changes the value to an newly initialized, empty object
C	Yes	Com Col	o.a.c.c.m.ListOrderedMapValuesView	size()	Yes	2	Return the size by calling the super.getSize() method
O	No	Com IO	o.a.c.i.f.TrueFileFilter	accept(File,String)	Yes	2	The method always returns true but this is implied by the class name and explained in the java doc. Further more, it only follows an interface. No way of getting around this name
C	Yes	Tapestry	o.a.t.c.Foreach	prepareForRender(IRequestCycle)	Yes	2	Method initializes some variables
C	Yes	Tapestry	o.a.t.s.i.LinkFactoryImpl	setRequestCycle(IRequestCycle)	Yes	2	Normal setter
C	Yes	Tapestry	o.a.t.v.AbstractNumericValidator	buildRangeMessage(IFormComp,Numb,Numb)	Yes	2	Dispatches build call to other build methods depending on the content of the passed in variables
O	No	Ant	o.a.t.a.f.ClassConstants	read()	Yes	1	Method does a lot but in essence It reads the complete stream with which the class got initialized and initializes all the constants

Continued on next page

Table D.2 Result of the review of the *Høst and Østvold method* using the *orgininal attribute rule set* – continued from previous page

G	Cor	Project	Class	Method	Appr	Cert	Reason
O	Yes	Xalan-J	o.a.x.x.c.u.TypeCheckError	toString()	No	2	Breaks the contract of toString(). Creates objects if not initialized while toString() should not change program state (according to the Java Language Spec)
O	Yes	Xalan-J	o.a.x.x.u.IntegerArray	toIntArray()	No	1	Also copies the internal int array to a new int array
C	Yes	Xerces-J	o.a.x.d.RangeImpl	selectNode(Node)	Yes	1	Sets internal variables of object to the selected node
O	No	Xalan-J	o.a.x.d.r.DTMNamedNodeMap	getLength()	Yes	1	A lazy initialized length getter. This is allowed because the map itself is unmodifiable
O	Yes	Xalan-J	o.a.x.s.EmptySerializer	setMediaType(String)	No	2	Nothing actually happens âĂę but then again, the class is called EmptySerializer
O	Yes	Xalan-J	o.a.x.u.XMLStringDefault	getChars(int,int,char,int)	No	1	Actually does a copy. Nothing is returned.
C	Yes	Xalan-J	o.a.x.a.FilterExprIterator	setInnerExpression(Expression)	Yes	2	Set a value on this object but also sets the parent on the object that is passed into the method
O	No	Xalan-J	o.a.x.d.XPathResultImpl	getStringValue()	Yes	1	This method checks whether the return type is a string. If not it will throw an exception otherwise it will try to return the value as a string.
C	Yes	ArgoUML	o.a.m.m.CoreHelperMDRImpl	removeTaggedValue(Object,String)	Yes	2	Method removes a value if it is existing
C	No	ArgoUML	o.a.m.m.ModelManagementHelperMDRImpl	getAllModelElementsOfKind(Object,Object)	No	2	Method does a lot. Tries to do a lookup of all the models of a certain type. Does much more things then a get should do
O	No	ArgoUML	o.a.u.d.s.u.UMLClassDiagram	getActionComposition()	Yes	1	Again lazy loaded variable
O	No	ArgoUML	o.a.u.d.s.u.UMLClassDiagram	getActionPackage()	Yes	1	Method returns a field. If the field is not initialized it will get loaded (laze load). ArgoUML seems to use a lot of laze loading
C	Yes	ArgoUML	o.a.u.SuffixFilter	accept(File)	Yes	2	Method return a boolean if the given file contains the right file extention
C	Yes	Castor	o.c.c.CacheAcquireException	printStackTrace(PrintWriter)	Yes	2	Method prints the whole stack trace of the exception
C	Yes	Castor	o.c.j.c.DataSourceDescriptor	getIdentity()	Yes	2	Just a normal getter
C	Yes	Castor	o.c.j.c.JdoConfDescriptor	getXMLName()	Yes	2	Just a normal getter
O	Yes	Castor	o.c.p.p.RelationCollection	toArray(array)	No	1	Method copies the given array into a new array of possible a different length
O	Yes	Castor	o.e.c.p.FieldMolder	setValue(Object,Object,ClassLoader)	No	2	A big setter, it is a very generic setter that can set almost all fields
O	Yes	JEdit	org.gjt.sp.jedit.jEdit	getViews()	No	1	Method creates a array of all the views. Apparently view is a linked list of views.
C	Yes	Hibernate	o.h.p.e.BasicEntityPropertyMapping	toColumns(String,String)	Yes	1	Basic conversion method to convert the given input to another format (in this case an String[])
C	Yes	JBoss	o.j.e.p.c.j.k.JDBCDB2IdentityValLocalCreateCommand	executeInsert(int,PrepStat,EntEnterprCon)	Yes	2	Executes the insert command on the given prepared statement

Continued on next page

Table D.2 Result of the review of the *Høst and Østvold method* using the *orgininal attribute rule set* – continued from previous page

G	Cor	Project	Class	Method	Appr	Cert	Reason
O	No	JBoss	o.j.i.r.m.s.SkeletonStrategyExceptionWriter	write(OutputStream,Object)	Yes	1	Writes the exception to a output stream. If a writer method is set it calls that instead of doing it itself
O	Yes	JBoss	o.j.m.l.BasicLoaderRepository	getResources(String,ClassLoader,List)	No	1	Bad method. It creates resource urls and puts it in a list. This list is not returned but is a method parameter
O	No	JBoss	o.j.u.Classes	getAttributeSetter(Class,String,Class)	Yes	1	Returns the setter of a certain property in a class
O	No	JBoss	o.j.u.MuBoolean	set(MuBoolean)	No	2	Set the new value and returns the old value
C	No	JBoss	o.j.u.MuCharacter	setValue(Object)	No	2	Methods has a generic Object as input variable and then checks the class does casts and calls specific values on this casted object to get the right value
C	No	JikesRVM	o.j.c.o.i.OPT_Instruction	getPureUses()	No	1	Always returns a new instantiated object
C	Yes	JikesRVM	o.j.c.o.i.OPT_Register	append(OPT_Register)	Yes	2	Appends the given element as next element in the list
C	Yes	JikesRVM	o.j.VM	write(char)	Yes	2	Low level routine to write a char to the console or sys.err.out
C	Yes	JRuby	o.j.a.BlockPassNode	accept(NodeVisitor)	Yes	2	An accept on a visitor
O	Yes	JRuby	o.j.e.o.X509Cert	set_issuer(IRubyObject)	No	2	Does not set anything on the current object but on a generator. Returns the parameter which was given to the method in the first place without changing it
C	Yes	JRuby	o.j.r.Frame	setCallingZSuper(boolean)	Yes	2	A normal setter
O	Yes	JRuby	o.j.u.WeakIdentityHashMap	get(int,Object)	No	2	Method actually does a find by calculating the hash and looking into the correct bucket. The thing is âĂę it needs to be called get because of the map interface
O	No	Spring	o.s.b.MutablePropertyValues	isEmpty()	Yes	2	Return whether or not a list is empty
O	Yes	Spring	o.s.j.e.a.AbstractConfigurableMBeanInfoAssembler	setNotificationInfos(array)	No	2	Converts the passed in array to an array of a different type before setting it
O	No	Spring	o.s.w.b.EscapedErrors	getGlobalErrors()	Yes	2	Returns a list of escaped errors
C	Yes	Spring	o.s.w.c.r.AbstractRequestAttributes	executeRequestDestructionCallbacks()	Yes	2	Calls a run method on all elements in its callback list
O	Yes	Spring	o.s.w.s.v.x.XsltView	getSourceTypes()	No	2	Creates a new array of the source types every time it is called.
C	Yes	TranQL	o.t.s.j.b.TimeBinding	getSQLType()	Yes	2	Return the type of the binding (which is a constant defined in java.sql.types)
O	No	TranQL	o.t.s.ReturnedValueTypeDetectorVisitor	visit(Query,Object)	Yes ?	1	It looks like a visit on a node but it is hard to understand the intention of the method

Table D.3: Result of the review of the *Høst and Østvold method* using the *narrowed percentiles set*

G	Cor	Project	Class	Method	Appr	Cert	Reason
O	Yes	JEdit	c.m.x.XmlParser	setAttribute(String,String,int,String,String,int)	No	1	Methods adds an attribute
C	No	FitNesse	f.r.RawContentResponderTest	getResultsUsing(String)	No	1	Method does more things then just returning a value, it changes state
C	Yes	FitNesse	f.r.r.ExecutionLog	addException(Throwable)	Yes	1	Adds the exception to the list
O	No	JBoss	j.m.MBeanNotificationInfo	toString()	Yes	1	Implementation of toString using StringBuffer
C	Yes	XML-Batik	o.a.b.a.s.JSVGViewerFrameUserAgent	getPixelToMM()	Yes	1	Delegates to another method
C	Yes	XML-Batik	o.a.b.c.d.CSSOMSVGColorAbstractComponent	getRectValue()	Yes	2	Method always throws exception when called but must be implemented because of interface
C	Yes	BSF	o.a.b.u.CodeBuffer	getConstructorExceptions()	Yes	1	Normal getter
C	Yes	Com Col	o.a.c.c.b.TreeBidiMap	isRed(TreeBidiMapNode,int)	Yes	1	Wrapper method
O	No	Com Col	o.a.c.c.m.CompositeMap	isEmpty()	Yes	1	Iterates over all the maps in its composition and returns false if one of these is not empty
O	No	Com Col	o.a.c.c.StaticBucketMap	size()	Yes	2	Gets the size by iterating over the different buckets
O	No	Com Http	o.a.c.h.HostConfiguration	getProtocol()	Yes	2	Delegated to other object in a null safe way to return the protocol
O	Yes	Com Lang	o.a.c.l.StringUtils	getLevenshteinDistance(String,String)	No	1	Method calculates the distance
C	Yes	Tapestry	o.a.t.e.DeferredScriptImpl	toString()	Yes	1	Normal toString
C	Yes	Tapestry	o.a.t.f.LabeledPropertySelectionModel	setOption(Object)	Yes	1	Normal setter
C	Yes	Ant	o.a.t.a.t.c.Equals	setTrim(boolean)	Yes	1	Normal setter
C	Yes	Ant	o.a.t.a.t.Input	setValidargs(String)	Yes	1	Normal setter
C	Yes	Ant	o.a.t.a.t.o.j.JJTree	setVisitorException(String)	Yes	2	Methods sets the passed in value into a map
O	Yes	Ant	o.a.t.a.t.o.NetRexxCTraceAttr	getValues()	No	1	Methods instantiates a new String[] everytime it is called
C	Yes	Xalan-J	o.a.x.t.ElemForEach	getTemplateMatch()	Yes	2	Returns this object
C	Yes	Xalan-J	o.a.x.x.d.NodeSortRecord	compareDocOrder(NodeSortRecord)	Yes	1	Return types an int
O	Yes	Xerces-J	o.a.x.i.d.x.XSSimpleTypeDecl	getActualValue(Object,ValCon,ValInf,boolean)	No	2	Very large method.
C	Yes	Xerces-J	o.a.x.j.v.ValidatorHandlerImpl	endPrefixMapping(String)	Yes	2	Wrapper method. But what does it end?
C	Yes	Xalan-J	o.a.x.d.r.s.SAX2DTM2TypedAttributeIterator	next()	Yes	2	Returns the int of the next node
C	Yes	ArgoUML	o.a.m.AbstractStateMachinesHelperDecorator	getAllPossibleSubvertices(Object)	Yes	1	Delegates to implementation
O	Yes	ArgoUML	o.a.p.XmlInputStream	isLastTag(int)	No	1	Method not only returns a boolean but also changes inner state of the object
O	No	ArgoUML	o.a.u.ProjectBrowser	clearDialogs()	Yes	1	Disposes all the windows that are owned by the object
C	Yes	ArgoUML	o.a.u.d.s.u.SelectionComment	createEdgeLeft(MutableGraphModel,Object)	Yes	2	Method actually does not create anything itself but delegates a call to another object (which is passed in as variable)
C	No	ArgoUML	o.a.u.r.u.RESequenceDiagramDialog	buildEdge(String,FigClassifierRole,FigClassifierRole,Object)	No	2	Not only the edge is build, a lot of operations are done in this method
O	Yes	ArgoUML	o.a.u.u.b.s.PropPanelTransition	getTriggerActions()	No	1	Method creates a new array and returns this
O	No	Castor	o.c.u.c.ConcurrentHashMap	clear()	Yes	1	Clears the underlying map implementation
C	Yes	Castor	o.e.c.b.d.DescriptorJClass	addClassDescriptorOverrides(boolean)	Yes	2	Code generator. Adds the @Override annotation and extend behavior

Continued on next page

Table D.3 Result of the review of the *Høst and Østvold method* using the *narrowed percentiles set* – continued from previous page

G	Cor	Project	Class	Method	Appr	Cert	Reason
C	Yes	JEdit	o.g.s.j.s.KeywordMap	getNonAlphaNumericChars()	Yes	1	Delegates to another object
O	Yes	Hibernate	o.h.e.JoinSequence	getFromPart()	No	1	Instantiates a new object and returns this
C	Yes	Hibernate	o.h.m.Collection	getOrderBy()	Yes	1	Normal getter
C	Yes	Hibernate	o.h.u.IdentityMap	toString()	Yes	1	Delegates toString to internal map
O	No	JBoss	o.j.e.p.CMPFilePersistenceManager	isStoreRequired(EntityEnterpriseContext)	Yes	1	Checks whether or not changes were made
O	No	JBoss	o.j.m.i.u.m.StreamDemux	setFrameSize(short)	Yes	1	Set the frame size in a synchronized way
O	No	JBoss	o.j.m.r.e.CompositeQueryExp	setMBeanServer(MBeanServer)	Yes	1	Set the server for all the underlying query exponents
O	Yes	JBoss	o.j.p.c.Proxies	getInvocationHandler(Object,Class)	No	2	Method only returns existing variable in some cases, otherwise it instantiates a new object
O	Yes	JBoss	o.j.s.a.s.UsernamePasswordLoginModule	getUsernameAndPassword()	No	2	Method gets the username and password by calling other callback functions after which the retrieving of the name and password continue. Bad return type btw
O	Yes	JBoss	o.j.s.s.j.SRPCacheLoginModule	getUserInfo()	No	2	Same as with the getUsernameAndPassword method
O	Yes	JBoss	o.j.s.ServiceDynamicMBeanSupport	getMBeanInfo()	No	1	Instantiates a lot of objects to return the info
C	Yes	JBoss	o.j.w.t.s.CustomPrincipalValveUserPrinicpalRequest	getDecodedRequestURI()	Yes	1	Delegated getter
C	Yes	JikesRVM	o.j.c.o.i.Call	getClearResult(OPT_Instruction)	Yes	2	Delegates to another object also does an assert
C	Yes	JikesRVM	o.j.c.o.i.OPT_Register	setVolatile()	Yes	2	Adds volatile as flag.
C	Yes	JikesRVM	o.j.c.o.OPT_Diamond	getTaken()	Yes	1	Normal getter
C	Yes	JRuby	o.j.a.ToAryNode	accept(NodeVisitor)	Yes	1	Clean implementation of visitor accepts method
O	Yes	JRuby	o.j.e.o.x.PEM	write_DSAPriKey(Writer,DSAPriKey,String,array)	?		Method does a lot
O	Yes	JRuby	o.j.e.o.x.X509_STORE_CTX	get1_issuer(array,X509AuxCertificate)	? No	1	Don't understand what happens
O	Yes	JRuby	o.j.u.Glob	getNames()	No	2	Method gets all the files and then iterates them for their names and only if they match a pattern.
O	Yes	ASM	o.o.a.ClassReader	accept(ClassVisitor,array,boolean)	No	2	Very large method (1000 lines). Does a lot of things. Actually looks like it does manually visits on members
O	No	Spring	o.s.j.s.r.ResultSetWrappingSqlRowSet	getRow()	Yes	1	Can throw exception when there is a problem with the SQL
C	Yes	Spring	o.s.o.h.LocalSessionFactoryBean	setLobHandler(LobHandler)	Yes	1	Normal setter
O	No	Spring	o.s.o.h.LocalDataSourceConnectionProvider	getConnection()	Yes	1	Can throw exception
O	No	Spring	o.s.t.i.DelegatingTransactionAttribute	equals(Object)	Yes	1	Delegates equals call. The class is called DelegatingTransactionAttribute
O	No	Spring	o.s.u.AutoPopulatingList	lastIndexOf(Object)	Yes	1	Delegated setter
O	No	Spring	o.s.v.BindException	getFieldError()	Yes	1	Delegated getter
O	No	Spring	o.s.w.p.u.PortletRequestWrapper	getParameterMap()	Yes	1	Delegated getter
C	Yes	TranQL	o.t.d.AbstractNode	isOnlyChild()	Yes	1	Normal check whether its an only child
C	Yes	Polyglot	p.t.Package_c	toType()	Yes	2	Method comes from inherited interface, always return null

Table D.4: Result of the review of the *FCA method*

G	Cor	Project	Class	Method	Appr	Cert	Reason
C	No	Ant	o.a.t.a.t.MacroInstance	addText(String)	No	1	Method does not add anything but just sets the text attribute
O	No	Ant	o.a.t.a.t.GenerateKey	createDname()	Yes	1	Method does some checks whether the object can get created. Otherwise throws a runtime exception.
O	No	Ant	o.a.t.a.t.o.Cab	createExec()	Yes	1	Method just instantiates a new object and returns this newly created instance
O	No	Ant	o.a.t.a.t.XSLTProcessParam	getName()	Yes	1	Method does a null check. If the field that will be returned is null it throws a runtime exception.
O	No	Antlr	a.BaseAST	addChild(AST)	Yes	1	Method adds the child to the existing child list or otherwise it adds it to a new list
C	Yes	ArgoUML	o.a.u.d.c.u.FigClassifierRole	getLineWidth()	Yes	1	Just a normal getter
O	Yes	ArgoUML	o.a.u.d.s.u.FigInstance	getMinimumSize()	No	2	Methods calculates the minimum size instead of just returning a value
C	No	ArgoUML	o.a.u.u.b.s.UMLTransitionTriggerList	getPopupMenu()	No	1	Method always instantiates a new object. Create would be a better name
O	No	ArgoUML	o.a.u.u.ActionGenerateAll	isEnabled()	Yes	1	Method just checks whether a certain condition is met
C	Yes	BCel	o.a.b.c.Field	accept(Visitor)	Yes	1	Method / class implements default Visitor pattern
C	Yes	BCel	o.a.b.c.AccessFlags	isStatic()	Yes	1	Methods checks whether the static flag is set
C	Yes	Castor	o.c.j.e.SQLTypeConverters	convert(Object,String)	Yes	1	Method(s) convert values from one type of boxed primitive to another
O	No	Castor	o.e.c.n.u.URILocationImpl	getReader()	Yes	2	Lazy loaded get. If the needed objects are not yet initialized then these are first initialized
C	No	Commons Collections	o.a.c.c.FastArrayList	addAll(int,Collection)	No	1	JavaDoc already has a give a way: Insert all of the elements in the specified Collection at the specified position in this list
O	No	Commons Collections	o.a.c.c.SequencedHashMap	getEntry(int)	Yes	1	Returns a value at a certain position in the Map
O	No	Commons Collections	o.a.c.c.l.AbstractLinkedListLinkedListIterator	getLastNodeReturned()	Yes	2	Return the node that is currently retained by the object. Nothing special except it can possibly throw an exception
O	Yes	FitNesse	f.r.t.PageHistoryResponderTest	addDummySuiteResult(File)	No	2	The method instantiates a lot of objects. Create would be better suited. But then again, this is a test method.
O	Yes	FitNesse	f.w.SymbolicPage	getData()	No	2	The method changes state on the object it returns before it actually returns it
O	Yes	FitNesse	f.r.r.f.CompositeFormatter	getErrorCount()	No	2	Methods iterates throw an object list and calls a getErrorCount method on these object. The highest result is returned

Continued on next page

Table D.4 Result of the review of the *FCA method* – continued from previous page

G	Cor	Project	Class	Method	Appr	Cert	Reason
O	Yes	Hibernate	o.h.m.DenormalizedTable	createForeignKeys()	No	1	There is a code smell here. The method is called create but does not return anything. Instead it iterates a list and calls another method called createForeignKeys which actually does return a key.
O	No	Hibernate	o.h.c.PersistentList	getOrphans(Serializable,String)	Yes	2	Method delegates to another static getOrphans() method
O	No	Hibernate	o.h.j.AbstractBatcher	getResultSet(CallableStatement,Dialect)	Yes	2	Method returns a result set but also add the result set to a list of result sets that need to be closed later
C	Yes	Hibernate	o.h.c.PersistentBag	toArray()	Yes	1	Before it converts the array to an array it does a lazy initialization
O	No	Hibernate	o.h.p.Printer	toString(array,array)	Yes	1	Method iterates over a collection and calls other to..String methods to collect the total string
O	No	JBoss	o.j.p.c.ProxyImplementationFactory	createConstructor()	Yes	1	Method creates a byte-code constructor implementation
O	Yes	JBoss	o.j.d.XSLSubDeployer	createService()	No	2	Method does not return the created service instance, it initializes it
O	Yes	JBoss	o.j.m.SpyXAException	getLinkedException()	No	1	Method always creates a new exception
C	Yes	JBoss	o.j.i.CorbaORBService	getSSLPort()	Yes	1	Just a normal getter
C	No	JBoss	o.j.m.i.o.OILServerIL	getTemporaryQueue(ConnectionToken)	No	1	Not only returns the temporary queue but also connects to the queue
O	No	JBoss	o.j.e.p.c.j.b.JDBCEntityBridge2	remove(EntityEnterpriseContext)	Yes	1	Removes the passed in context from all underlying fields
O	Yes	JBoss	o.j.h.j.HANamingService	setClientSocketFactory(String)	No	2	Method instantiates a new factory class. The parameter passed in is only used to set the name of the factory class
O	Yes	JBoss	o.j.s.Main	setServerSocketFactory(String)	No	2	Same as the setClientSocketFactory(). Method uses passed in variable to instantiate a new factory using threadlocal classloader
C	Yes	JEdit	o.g.s.j.g.VariableGridLayout	getLayoutAlignmentY(Container)	Yes	2	Always returns 0.5f. Methods needs to have this name because of implemented interface. The improvement could be here not to return the 0.5f value but to return a constant
O	No	JEdit	o.g.s.u.PropertiesBean	getPropertyDescriptors()	Yes	1	Result is delegated to other object
C	Yes	JHotDraw	C.i.d.c.h.DiamondFigureGeometricAdapter	getShape()	Yes	1	Method delegates to another method (which is inherited)
O	No	JikesRVM	o.j.c.o.i.OPT_AssemblerBase	getDisp(OPT_Operand)	Yes	2	Return a value but needs a type cast for it
C	Yes	JikesRVM	o.j.c.o.i.OPT_AssemblerBase	getIndex(OPT_Operand)	Yes	2	Same as the getDisp method
O	No	JikesRVM	o.j.c.o.OPT_AnnotatedLSTNode	getMonotonicStrideValue()	Yes	2	Returns the value depending on the internal state of an object
C	No	JikesRVM	o.j.c.o.i.MIR_CondBranch2	indexOfBranchProfile1(OPT_Instruction)	No	2	Method always returns the value 2 but next to that can potential fail depending on the passed in method variable

Continued on next page

Table D.4 Result of the review of the *FCA method* – continued from previous page

G	Cor	Project	Class	Method	Appr	Cert	Reason
C	Yes	JRuby	o.j.e.o.SSLContext	initialize(array)	Yes	2	Initializes the internal state of the object and returns itself. Strange thing: nothing is done with the passed in variable
C	Yes	JRuby	o.j.a.v.AbstractVisitor	visitOptNNode(OptNNode)	Yes	1	Method delegates to an general visit method
C	No	Spring	o.s.w.s.m.BaseCommandController	checkCommand(Object)	No	2	Method name is very general but the implementation is very concrete. The method should be called something like: isCommandCompatible
C	Yes	Spring	o.s.c.s.DefaultMessageSourceResolvable	equals(Object)	Yes	1	Normal equals. Uses an external object to do a null safe equals on the internal field
C	Yes	Spring	o.s.w.p.HandlerExecutionChain	getHandler()	Yes	1	Normal getter
C	Yes	Spring	o.s.b.f.c.ConstructorArgumentValuesValueHolder	setValue(Object)	Yes	1	Normal setter
O	Yes	Struts	o.a.s.c.t.WebTable	setSortable(boolean)	No	2	Not only changes the internal boolean value but also changes, if needed, the implementation of the internal model
C	Yes	Tapestry	o.a.t.a.DefaultAssetFactory	createAsset(Resource,String,Locale,Location)	Yes	2	Creates a new ExternalAsset and returns this object. Strange thing, there are more parameters passed into this method than there are used
C	Yes	Tapestry	o.a.t.s.SpecFactory	createBindingBeanInitializer(BindingSource)	Yes	1	Creates and returns an object
O	Yes	Tapestry	o.a.t.f.LabeledPropertySelectionModel	getValue(int)	No	2	Conditional get? Depending on the value of the passed in index a result value is returned
C	Yes	Tapestry	o.a.t.s.ExtensionSpecification	toString()	Yes	1	Normal toString using a StringBuffer
O	Yes	TranQL	o.t.e.ManyToManyCMR	get(InTxCache,CacheRow)	No	1	Creates a new object and returns this
C	No	Xalan-J	o.a.x.x.t.SmartTransformerFactoryImpl	newXMLFilter(Source)	No	2	The name new is used but it should have been create
O	No	Xalan-J	o.a.x.d.r.DTMStringPool	removeAllElements()	Yes	1	Removes all elements from the internal lists
C	Yes	Xerces-J	o.a.x.i.x.XMLSchemaValidatorXPathMatcherStack	addMatcher(XPathMatcher)	Yes	1	Add the passed in matcher into the existing array. Array gets increased if the size is not big enough
C	No	Xerces-J	o.a.x.i.x.t.XSDAttributeTraverser	checkDefaultValid(XSAttributeUseImpl)	No	2	Method validates default values. Should have been called validate
C	Yes	Xerces-J	o.a.x.d.CoreDocumentImpl	getErrorChecking()	Yes	1	Normal getter
O	No	Xerces-J	o.a.x.d.NodeImpl	getNodeNumber()	Yes	2	Method delegates call
C	Yes	Xerces-J	o.a.h.d.HTMLLinkElementImpl	getRel()	Yes	2	Method delegates call
C	Yes	XML-Batik	o.a.b.e.a.i.r.SpecularLightingRed	copyData(WritableRaster)	Yes	1	Copies it data onto the passed in object
C	Yes	XML-Batik	o.a.b.s.s.JSVGComponentBridgeUserAgentWrapperQuery	run()	Yes	2	Anonymous inner class with implementation of run. Nothing strange is happening there