Discovering Strategies for Method Naming

Jeroen Peeters
The University of Amsterdam

November 9, 2015

Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt
— Ludwig Wittgenstein —

Master Thesis
Software Engineering

Supervisor: dr. Alexander Serebrenik
dr. Hans Dekkers

— Table of Contents —

1 Introduction

1.1 What's in a name

Naming pieces of code is one of the most important tasks of a software developer. Giving a piece of code the correct name is important for understandability. But what is the right name? Providing good quality names is generally believed to be of high importance. Though there may be some guidelines, no exact or mechanical process exists to name pieces of code. This thesis seeks to discover the strategies people use to create names for existing pieces of code written in the Java programming language.

1.1.1 Names in the Java programming language

Before we continue we should understand which nameable parts exist the Java programming language. In the Java programming language the following constructs are nameable.[2]

- Classes
- Methods
- Variables
- Method arguments
- Return types

Examples of the above constructs are listed in appendix A.

1.2 What's in a good name?

For computers to interpret and execute program code there is no need for it to be laid out or indented. Just the same the computer doesn't care about nice, well understandable and self-explanatory names. We do this to communicate the intent of the code to other developers or even our future self should we come back to extend the code or fix bugs.

1.2.1 Name characteristics

Characteristics of names in programming:

- Reveal intentions
- Length
- One word per concept
- Use names from the domain

1.3 Characteristics in code

Zoals besproken op Skype:

De huidige opzet van mijn scriptie/onderzoek blijkt lastig te voltooien. Jij stelde het volgende voor; ik kom een week naar de UvA om studenten en misschien docenten te interviewen over de vraag hoe mensen komen tot de naamgeving van een methode, gegeven de implementatie en context. Deze kennis kan interessant zijn om code automatisch te labelen bij automatische refactoring. Jij merkte op dat het wellicht interessant is om uit te zoeken welke karakteristieken in de code nu echt belangrijk zijn voor het geven van een naam. Het idee is om code met veel verschillende karakteristieken te geven, een complex vraagstuk. Volgens Kahnemen zal dit leiden tot een substituut vraag, welke karakteristieken zijn nu nog belangrijk? Altijd dezelfde? Of altijd de eerste N die men tegenkomt bij het lezen van de code. Maakt de context uberhaupt uit? of is alleen de implementatie genoeg? of is misschien juist alleen de context genoeg voor het geven van een naam? Worden andere namen gegeven bij afwezigheid van context of implementatie. Verder kan nog worden gekeken naar de strategie die mensen kiezen: beantwoorden ze elke vraag op eenzelfde manier waardoor de gegeven namen ook vergelijkbaar zijn opgebouwd. Mogelijkheid is ook dat mensen geen naam kunnen geven.

Jij noemde ook inconsistentie als mogelijkheid om naar te kijken en je noemde als voorbeeld het gebruik van naamgevingsconventies; zou je nog kort kunnen toelichten wat je hiermee bedoelde?, ik kon dit uit m'n aantekeningen niet direct meer opmaken :S

Het idee is, volgens mij, om het volgende te organiseren: over 3-4 weken een week op de Uva voor het afnemen van interviews. Ik bereid vragen voor op papier met random vragen uit een codebase van github van meestgebruikte open-source software. Interview is interactief volgens 'thinking aloud protocol'.

Deze week + weekend zal ik dit plan verder uitwerken + voorbeeld vragen voorbereiden die wij maandag 9 november kunnen bespreken. Ik zal dan in de middag, na de lunch (rond 13:00u?), staan bij C302.4.

Hans, hartelijk dank voor je tijd zover en je aanbieding om studenten te interviewen.

Groet, Jeroen

2 Research Question

- Can people reverse-engineer method names?
  Given a method implementation and contextual information, can people reconstruct the intent of that method and give it an appropriate name?

  What is the minimal required information?

2.1 Hypothesis

The main driver of the hypotheses is the intent of the code. The intent of the code primarily exhibits in the names. Therefore the hypotheses are grouped per characteristic (1.2.1).

2.1.1 Reveal intentions

1. When the code is simple and uses coherent names, comments will be ignored.
2. Smaller methods, such as one liners, are missing context. Without this context it will be impossible to get the intent and thus provide an appropriate name.

2.1.2 Length

2.1.3 One word per concept

1. When the code exhibits obvious and coherent names these names will be re-used in the method name.

2.1.4 Use names from the domain

1. People without domain knowledge will not be able to grasp the intent of the code. They will rely more on context, such as comments, than what the code actually expresses.
2. People without domain knowledge will have more difficulties naming larger and complex methods.

1. Without providing context, simple code samples such as one liners, will be hard to name.
2. Names given to pieces of code obtained by Extract-Method Refactoring are of better quality because it is a process the developer is familiar with.
3. When the code exhibits common patterns with which the developer is familiar, he will answer the question from experience.
   [BETER VERWOORDEN]

Hypothesis	*Parameter variation*			
	Method Javadoc	Method call example	C	D
H1	X1	X2	X3	X4
H2	Y1	Y2	Y3	Y4
H3	Y1	Y2	Y3	Y4
H4	Y1	Y2	Y3	Y4
H5	Y1	Y2	Y3	Y4
H6	Y1	Y2	Y3	Y4

Matrix met parameters

toon classes, documentatie, grote methode. welke parameters ga ik varieren. mensen die commentaar niet gebruiken, ervaring met onbruikbaar commentaar.

Frank Nack. Sander Bakkes. -¿ UXlab

3 Research Method

3.1 Semi structured interviews

The goal is to obtain data about how people reverse engineer method names from code. To try to answer this question I've held interviewing sessions with respondents from the field of software engineering. The group of respondents exists of students from the Software Engineering Master at the University of Amsterdam and professional software engineers from ICTU[1]. For research purposes generally two types of interviews are employed, structured and unstructured interviews.

In structured interview questions are asked from a standardized set of which the order is known. Generally more closed questions are used which makes the interview easier to replicate because they generate quantitative data. On the other hand, it is not possible to ask about motives or reason behind an answer.

An unstructured or informal interview on the other hand acts as a conversation guide. Generally more open-ended questions are used which give the opportunity to tailor the follow-up questions to the answers of the respondent. It gives the interviewer the possibility to validate answers, get deeper understanding and ask for the rationale behind an answer. This type of research is harder to replicate as it generates more qualitative data.

In my interviews I use a predefined list of code samples and questions. I will use open questions to get the conversation started. The idea is that an open question will trigger the respondent to think deeper about the code sample. While engaging in a conversation, I use a list of predefined closed questions to test for validity and ask for an explanation and rationale.

3.2 Varying code characteristics

I've created created three variants of interviews that use the same code samples with varying degrees of contextual information. In ordre to discover which pieces of information are of most importance I will vary:

- JavaDoc of a method implementation
- Method call example
- .....

... TBD; section about how I vary context for the code samples.

[1] ICTU (http://www.ictu.nl/), semi governmental agency for IT projects

4 Verification model

4.1 Survey: Naming Java Methods

Naming pieces of code is one of the most important tasks of a software developer. Giving a piece of code the correct name is important for understandability. But what is a correct name? Providing good quality names is generally believed to be of high importance. Though there may be some guidelines, no exact or mechanical process exists to name pieces of code. In order to understand how developers create method names, I've executed a survey among a group of peers.

4.1.1 Survey setup

The survey is executed as an online questionnaire which is filled out individually without direct supervision. Participants are asked to provide names for shown nameless Java methods. Together with the method implementation limited contextual information is given, such as the name of the containing class and examples of how the method is used. Because the number of methods that can be named by each participant can vary greatly there's no maximum number of questions. Instead, there's a time limit of thirty minutes per participant. After this time, the questionnaire will stop automatically. At any moment, the participant can pause and resume the questionnaire at a later time.

4.1.2 Method corpus

In order to make sure that the methods included in the survey are a representative sample I've employed the following strategy. The methods used in the survey are taken from open-source Java projects with a high usage frequency. In other words, which projects are depended upon the most? See appendix D for detailed information on how the list was compiled.

- JUnit
- Log4J
- Commons IO
- Guava
- Commons-lang
- Mockito

In order to make sure that the methods taken from these projects are also representative I use the SIG maintainability model [1]. To make sure that any degree of small & large and simple & complex method are selected I use the following two properties:

- Complexity per unit
  The complexity of source code units inuences the systems changeability and its testability.
- Unit size
  The size of units inuences their analysability and testability and therefore of the system as a whole.

4.1.3 Results

4.2 Deducing the model

5 Acknowledgement

TBD...

References

[1] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[2] Oracle. Java language and virtual machine specification, 2015. `https://docs.oracle.com/javase/specs/`.

Some Appendix

Appendix A Names in the Java programming language

```
public class MyClass extends OtherClass {
    ...
}
```

Listing 1: Class names

```
public class _ {
    public void myMethodName(){
        ...
    }
}
```

Listing 2: Methods

```
public class _ {
    private int myClassVariable;

    public void ...(){
        String myLocalVariable = "some value";
    }
}
```

Listing 3: Variables

```
public class _ {
    public void _(int myArgument1, String myArgument2){
        ...
    }
}
```

Listing 4: Method arguments

```
public class _ {
    public String _(){
        ...
    }
    public int _(){
        ...
    }
}
```

Listing 5: Return type

Appendix B Interviews

B.1 Interview A

```
public class FilenameUtils {
    ...
    /**
    * Determines if Windows file system is in use.
    *
    * @return true if the system is Windows
    */
    static boolean ...() {
        return SYSTEM_SEPARATOR == WINDOWS_SEPARATOR;
    }
    ...
}
```

Listing 6: Commons IO 2.4 FilenameUtils.isSystemWindows Variant 1

```
public class FilenameUtils {
    ...
    static boolean ...() {
        return SYSTEM_SEPARATOR == WINDOWS_SEPARATOR;
    }
    ...
}
```

Listing 7: Commons IO 2.4 FilenameUtils.isSystemWindows Variant 2

```
public class IOUtils {
    ...
    /**
     * Compare the contents of two Readers to determine if they are equal or
     * not, ignoring EOL characters.
     * <p>
     * This method buffers the input internally using
     * <code>BufferedReader</code> if they are not already buffered.
     *
     * @param input1  the first reader
     * @param input2  the second reader
     * @return true if the content of the readers are equal (ignoring EOL differences),
     *    false otherwise
     * @throws NullPointerException if either input is null
     * @throws IOException if an I/O error occurs
     * @since 2.2
     */
    public static boolean ...(Reader input1, Reader input2)
            throws IOException {
```

```
20          BufferedReader br1 = toBufferedReader(input1);
21          BufferedReader br2 = toBufferedReader(input2);
22
23          String line1 = br1.readLine();
24          String line2 = br2.readLine();
25          while (line1 != null && line2 != null && line1.equals(line2)) {
26              line1 = br1.readLine();
27              line2 = br2.readLine();
28          }
29          return line1 == null ? line2 == null ? true : false : line1.equals(line2);
30      }
31      ...
32  }
```

Listing 8: Commons IO 2.4 IOUtils.contentEqualsIgnoreEOL Variant 1

```
1   public class IOUtils {
2       ...
3       public static boolean ...(Reader input1, Reader input2)
4               throws IOException {
5           BufferedReader br1 = toBufferedReader(input1);
6           BufferedReader br2 = toBufferedReader(input2);
7
8           String line1 = br1.readLine();
9           String line2 = br2.readLine();
10          while (line1 != null && line2 != null && line1.equals(line2)) {
11              line1 = br1.readLine();
12              line2 = br2.readLine();
13          }
14          return line1 == null ? line2 == null ? true : false : line1.equals(line2);
15      }
16      ...
17  }
```

Listing 9: Commons IO 2.4 IOUtils.contentEqualsIgnoreEOL Variant 2

```
public class Monitor {
/**
   * Enters this monitor. Blocks at most the given time.
   *
   * @return whether the monitor was entered
   */
  public boolean _(long time, TimeUnit unit) {
    long timeoutNanos = unit.toNanos(time);
    final ReentrantLock lock = this.lock;
    if (!fair && lock.tryLock()) {
      return true;
    }
    long deadline = System.nanoTime() + timeoutNanos;
    boolean interrupted = Thread.interrupted();
    try {
      while (true) {
        try {
          return lock.tryLock(timeoutNanos, TimeUnit.NANOSECONDS);
        } catch (InterruptedException interrupt) {
          interrupted = true;
          timeoutNanos = deadline - System.nanoTime();
        }
      }
    } finally {
      if (interrupted) {
        Thread.currentThread().interrupt();
      }
    }
  }
}

// Usage example:
monitor._(5, TimeUnit.SECONDS);
try {
    // do things while occupying the monitor
} finally {
    monitor.leave();
}
```

Listing 10: Guava 17.0 Monitor.enter Variant 1

```
public class Monitor {
  public boolean _(long time, TimeUnit unit) {
    long timeoutNanos = unit.toNanos(time);
    final ReentrantLock lock = this.lock;
    if (!fair && lock.tryLock()) {
      return true;
    }
    long deadline = System.nanoTime() + timeoutNanos;
    boolean interrupted = Thread.interrupted();
    try {
      while (true) {
        try {
          return lock.tryLock(timeoutNanos, TimeUnit.NANOSECONDS);
        } catch (InterruptedException interrupt) {
          interrupted = true;
          timeoutNanos = deadline - System.nanoTime();
        }
      }
    } finally {
      if (interrupted) {
        Thread.currentThread().interrupt();
      }
    }
  }
}
```

Listing 11: Guava 17.0 Monitor.enter Variant 2

B.2 Questions to ask during the interview

B.2.1 General questions before the interview

1. Do you have practical experience with the Java programming language?
2. If yes, how many years of experience do you have?
3. With which other languages do you have practical experience?

B.2.2 Questions to ask per code sample

1. Do you have experience with ****domain of code sample****
2. Do you understand and can you explain what this code does?

Appendix C Interview script

1. Please provide a name for the method with the missing name.
2. When unable
   (a) ask: Do you have experience/are familiar with ****domain of code sample****
   (b) When negative: explain/provide reading material about the domain.
   (c) Please provide a name for the method with the missing name.
   (d) When unable, ask: Do you understand/can you explain what this code does?
   (e) When negative: give more contextual information
   (f) Repeat last two steps until all contextual information is provided.
3. When able
   (a) ask: Can you explain why you chose this particular name and how did you construct it?
   (b) ask: Do you have experience with ****domain of code sample****
   (c) Do you understand and can you explain what this code does?

Appendix D Frequency of Java Dependencies

This is an edited version of my original article as posted on my personal website: http://www.jeroenpeeters.nl/articles/frequency-of-java-dependencies/.

D.1 The Approach

Github exposes an API through which you can search for projects with a certain language, rating, etc. Furthermore it is possible to query a projects tree structure and obtain file data.

Because I needed a representative data set I choose to include mature and active projects only. For this purpose a project is considered active if it had at least one commit in the last year. Secondly a project is considered mature if it is older than at least one year.

The Java world has three major build and dependency management tools; Maven, Gradle and Ivy. I simply downloaded the according build files to obtain dependency related information.

For each Github project each dependency is only counted once. This means that if a project contains multiple modules each having its own dependency management, duplicated dependencies are counted as one occurrence. Furthermore, build tools specific dependencies (such as maven-compiler-plugin) are omitted from the results. This is because depending on these artifacts is a consequence of using the build tool. They would thus occur frequently and cloud the results.

D.2 The Results

From the years 2008 to 2012 I was able to retrieve 3.029 projects with dependency management files (of which 2502 (82%) Maven projects, 430 (14%) Gradle projects and 97 (3%) Ant+Ivy projects).

From these figures it is not difficult to conclude that the majority of Java projects use Maven as a build and dependency management tool.

These projects had a total of 26.235 unique dependencies. The following list details the top 5 projects on which others depend:

- junit - 1883
- slf4j-api - 764
- oss-parent - 700
- log4j - 671
- commons-io - 543

The following graphic shows the top 25 most depended on projects. We observe that JUnit is by far the most depended on artifact, followed by sfl4j-api and oss-parent.

We can see that a large portion of these top projects are related to testing (junit, mockito-all, spring-test, mockito-core) and logging (sfl4j-api, log4j, slf4j-log4j12, commong-logging, logback-classic).

D.3 The code

To obtain and analyze the Github data I had to implement two relatively small programs. Both of them are freely available under the GNU GPL from, off course, Github.

- https://github.com/jeroenpeeters/github-dependency-analyzer

D.4 The data

The full data set contains 25.243 projects. Table 1 lists the first 30 projects from the data set with their dependency frequency. The complete data set, in raw and analyzed form, can be downloaded from the above mentioned Github repository.

Project name	**Dependency count**
junit	1883
slf4j-api	764
oss-parent	700
log4j	671
commons-io	543
guava	520
servlet-api	489
slf4j-log4j12	482
commons-lang	444
commons-logging	436
mockito-all	385
commons-codec	335
lifecycle-mapping	333
spring-context	332
httpclient	290
spring-test	288
logback-classic	284
joda-time	283
jackson-mapper-asl	273
jcl-over-slf4j	264
testng	263
spring-core	263
mockito-core	261
android	244
spring-beans	235
spring-web	234
commons-collections	228
hsqldb	218
spring-webmvc	216
mysql-connector-java	215

Table 1: Dependency frequency: Thirty most depended-on projects