On Method Ordering

Thesis submitted in partial fulfillment of the requirements for
the M.Sc.
degree in the
School of Computer Science, Tel-Aviv University

by

Yorai Geffen

This work has been carried out under the supervision of
Dr. Shahar Maoz

April 2016
Acknowledgments

We thank Nimrod Busany for advice on the analysis of our results. We also thank all the pilot participants of our online questionnaire.
Abstract

As the order of methods in a Java class has no effect on its semantics, an engineer can choose any order she prefers. Which code conventions regarding methods ordering are common in practice, if any? Are some orders better than others in making the code easier to understand? Can good orders be computed and applied automatically?

In this work we address these questions. First, we present a study of method orders in a large body of open source projects, where we identify existing common practices. Second, we present four method ordering strategies, which we automate and provide in an Eclipse plugin, as a form of refactoring. Finally, we present the results of a user study, which evaluates the effect of our methods ordering strategies on engineers’ code comprehension in terms of correctness and time spent on answering.
Contents

1 Introduction 1

2 Method Ordering Strategies 3
   2.1 StyleCop (SC) and Relaxed-StyleCop (SC-R) 3
   2.2 Calling (CL) 5
   2.3 Connectivity (CT) 6
   2.4 Calling+Connectivity (CLCT) 8

3 Method Ordering Strategies in Practice 9
   3.1 Scoring Functions 9
   3.2 Empirical Study 12
   3.3 Results and Observations 12
   3.4 Threats to Validity 14

4 User Study 17
   4.1 Experiment Setup and Execution 17
   4.2 Results and Conclusions 19
   4.3 Threats to Validity 24

5 Prototype Tool: Ordi 27

6 Related Work 29
   6.1 Ordering 29
   6.2 Identifier Names, Length, and Style 31
   6.3 Linguistic Antipatterns 33
   6.4 Eye Tracking 35
   6.5 Code Regularity 36
   6.6 How to Measure Program Comprehension? 37
Chapter 1

Introduction

As the order of methods in a class has no effect on its semantics, an engineer can choose any order she prefers. Indeed, in the literature and available tools, one can find some related informal conventions and advice.

On the one hand, a well-known ordering convention requires that all public methods appear before all private ones. For example, the popular StyleCop tool [26] enforces a method ordering convention that includes this requirement. On the other hand, the Oracle (previously Sun) Java conventions document [20] prescribes that “methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods...”. As another example, the popular Clean Code book [19] suggests that “In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling. This creates a nice flow down the source code module from high level to low level...” [19, Chap. 5]. Thus, the freedom to choose the order on the one hand, and the different advice regarding the correct or best order on the other hand, call for investigation.

Which code conventions regarding methods ordering are common in practice, if any? Are some orders better than others in making the code easier to understand? Can good orders be computed and applied automatically, as a form of refactoring?

In this work we address these questions. First, we present four method ordering strategies (which we automate and provide in a tool, see below). We formally define these orderings and discuss the motivation to use them. One ordering strategy is the StyleCop strategy, which is based on method’s
CHAPTER 1. INTRODUCTION

modifiers. Another is the “Calling” strategy, where the caller methods appear before the callee. Another is the “Connectivity” strategy, where methods that are connected to each other (via method calls) appear adjacent to each other. The last strategy is the combination of the Calling and Connectivity strategies. We present the four ordering strategies in detail in Chapter 2.

Second, we present a study of method orders in a large body of open source projects, in view of the four strategies presented earlier, where we identify existing common practices. Our study shows that a method order derived from Clean Code’s ordering advice is common amongst these projects. We present the details of this study in Chapter 3.

Third, we present the results of a user study we have conducted, which evaluates the effect of our methods ordering strategies on engineers’ code comprehension. Our findings show that while method ordering has no clear effect on the correctness of comprehension, it does seem to have an effect on the time engineers spent in trying to answer code comprehension questions. We discuss the user study and its results in Chapter 4.

Finally, we have implemented our work and made it available in an Eclipse plugin named Ordi. Ordi can be used by managers and engineers to examine the conformance of a project, a package, or a single class to a method ordering strategy, and, most importantly, to reorder a class’s methods according to a selected strategy, as a form of refactoring.

All the data we report on in this Thesis, corpus analysis results and raw data of the user study, as well as a link to download the tool Ordi, is available from [http://smlab.cs.tau.ac.il/ordi/](http://smlab.cs.tau.ac.il/ordi/).

The results we report on in this Thesis will appear in an ICPC’16 paper [9].

The remainder of the Thesis is organized as follows. Chapter 2 presents the four ordering strategies. Chapter 3 presents our study on how engineers order methods in practice. Chapter 4 presents the user study we have conducted. The tool Ordi is described in Chapter 5. Chapter 6 discusses related work. Finally, Chapter 7 concludes and suggests future research.
Chapter 2
Method Ordering Strategies

Definition 1 (Method Ordering Strategy). A method ordering strategy $s$ defines a partial order on the methods in a given class. Specifically, given method $m_1$ and method $m_2$ of class $C$, the strategy defines whether $m_1 <_s m_2$ ($m_1$ should appear before $m_2$), $m_1 >_s m_2$ ($m_1$ should appear after $m_2$), or the order between them is undefined.

Based on preliminary literature review and manual exploration of many classes from different projects we present the following ordering strategies:

1. StyleCop (SC), based on method modifiers;
2. Calling (CL), based on the static call-graph of the class;
3. Connectivity (CT), based on connected components in the static call-graph of the class; and
4. Calling+Connectivity (CLCT), which combines CL and CT.

Below we motivate, define, and briefly discuss each of the four strategies. To demonstrate, we use the example class shown in List. 2.1.

2.1 StyleCop (SC) and Relaxed-StyleCop (SC-R)

The StyleCop (SC) strategy has specific rules for ordering methods inside a class, based hierarchically on the methods’ access and non-access modifiers. Specifically, methods should be grouped by their access modifiers, public, default, protected, and private, in this order. Within each of
CHAPTER 2. METHOD ORDERING STRATEGIES

Listing 2.1: Example class Java code

```java
public class OrdExampleClass {

    public OrdExampleClass(int x) {...}
    public OrdExampleClass(int y, int j) {...}

    public static void exampleMethod() {...}
    public void realExampleMethod() {
        exampleMethod();
    }

    private void doThis(int x) {
        doThat(x);
    }

    private void doNothing() {...}
    private static void doThat(int x) {...}
}
```

these groups, methods are divided into two subgroups, such that all static
methods should come before all non-static methods. Finally, within each
subgroup, methods are sorted alphabetically by their name. Methods with
the same name are ordered by ascending number of arguments. Constructors
are grouped first, ordered within them according to access modifiers etc.
The order between methods with identical modifiers, name, and number of
arguments, is not defined.

List. 2.2 shows the code of our example class, ordered according to the
SC strategy.

The rationale behind the SC ordering strategy is that it makes it easy
to quickly find a method implementation in the class, based on its access
modifiers and its name. In addition, grouping all public methods together
may help an engineer who is interested just in using the class API (and not
in understanding how it is implemented). Thus, it fits well with supporting
encapsulation.

The StyleCop tool [25, 26] uses the SC ordering strategy to enforce an
ordering convention.

In our work we consider the original StyleCop strategy described above
as well as a relaxed-StyleCop strategy (SC-R), where only the access and
static modifiers matter. Specifically, SC-R prescribes that all methods should
be grouped by their access modifiers, public, default, protected, and private, in this order (and static before non-static within each group). Constructors are grouped first, ordered within them according to access modifiers. In SC-R the order is oblivious to method names and their number of arguments.

Note that since SC-R is a relaxed version of SC, the code in Listing 2.2, which conforms to SC (gets a perfect score), conforms to SC-R as well.

### 2.2 Calling (CL)

The Calling (CL) strategy requires that when a method invokes another method, the invoking method should come before the invoked method. This means that when an engineer is searching for the implementation of the invoked method, she should only look further down, after the invoking method. The CL strategy is inspired by the advice of the Clean Code book [19], as cited in the introduction: “...a function that is called should be below a function that does the calling...” [19] Chap. 5.

Many tools and IDEs allow the engineer to automatically create a new method implementation right after the invoking method. This follows the

```java
public class OrdExampleClass {

    public OrdExampleClass(int x) {...}
    public OrdExampleClass(int y, int j) {...}

    public static void exampleMethod() {...}
    public void realExampleMethod() {
        exampleMethod();
    }
    private static void doThat(int x) {...}
    private void doThis(int x) {
        doThat(x);
    }
    private void doNothing() {...}

Listing 2.2: Example class ordered using the SC strategy
```
rationale of the CL strategy. Interestingly, the CL strategy goes against the rules of other programming languages, where procedures have to be defined before their use, not after, in order to allow efficient single-pass compilation.

To implement the CL strategy, we build a static call-graph of the methods in the class. In the graph, nodes represent methods. A directed edge between method \( m_1 \) and method \( m_2 \) means that \( m_1 \) may call \( m_2 \). The partial order for the CL strategy is defined by the call-graph, with the following two notes: we put the constructors first (technically by adding edges from all constructors to all other methods) and when methods are on a directed cycle, we consider the order between them to be undefined by the strategy.

List. 2.3 shows the code of our example class, ordered according to the CL strategy. Note, e.g., that the method \( \text{exampleMethod} \) appears after the method \( \text{realExampleMethod} \) because the latter calls the former.

### 2.3 Connectivity (CT)

The Connectivity (CT) strategy has one intuition in mind: related methods should be close to one another. This means that if a method causes another method to be invoked (does not have to invoke it itself), the two methods
2.3. CONNECTIVITY (CT)

```java
public class OrdExampleClass {

    public OrdExampleClass(int y, int j) {...}
    public OrdExampleClass(int x) {...}

    public static void exampleMethod() {...}
    public void realExampleMethod() {
        exampleMethod();
    }
    private void doThis(int x) {
        doThat(x);
    }
    private static void doThat(int x) {...}
    private void doNothing() {...}
}
```

Listing 2.4: Example class ordered using the CT strategy

should appear close to one another in the class code. The Connectivity strategy is inspired by Oracle (previously Sun) Java conventions document [20], which prescribes that “methods should be grouped by functionality rather than by scope or accessibility.”

The motivation behind this strategy is that when a developer wants to read the implementation of an invoked method, she should not look too far to find it, specifically, she should not have to cross over methods that are not related to the two methods at hand. The goal is to reduce context switching and noise.

To implement it, we build the same static call-graph of the methods in the class, as described above for the CL strategy, but then ignore edges’ directions and compute connected components. The partial order for the CT strategy is defined by the connected components in this undirected graph, with the exceptions of putting the constructors as one group. The order between methods that belong to the same connected component is undefined by the strategy.

List. 2.4 shows the code of our example class, ordered according to the CT strategy. Note that the two methods `exampleMethod` and `realExampleMethod` are grouped together and the two methods `doThis` and `doThat` are grouped together.
2.4 Calling+Connectivity (CLCT)

The CL and CT strategies defined above originate from a similar intuition. Still, both allow orders that may be counter productive. Specifically, a class may conform to the CL strategy although related methods are very far away (e.g., if a method located at the beginning of the class invokes a method which is placed at the very end of the class, with many other methods in between). Moreover, a class may conform to the CT strategy although it includes a large set of methods that are connected and thus the order between them is arbitrary.

To overcome the weaknesses of the CL and CT strategies, when considered alone, we propose the Calling+Connectivity (CLCT) strategy, which combines them. The strategy requires that connected methods are grouped together (as in the CT strategy) and that within each group, invoked methods come after invoking methods (as in the CL strategy).

List. 2.5 shows the code of our example class, ordered according to the CLCT strategy.

Listing 2.5: Example class ordered using the CLCT strategy

```java
public class OrdExampleClass {
    public OrdExampleClass (int x) {...}
    public OrdExampleClass (int y, int j) {...}

    public void realExampleMethod () {
        exampleMethod();
    }
    public static void exampleMethod () {...}
    private void doThis (int x) {
        doThat(x);
    }
    private static void doThat (int x) {...}
    private void doNothing () {...}
}
```
Chapter 3

Method Ordering Strategies in Practice

We now examine how engineers order methods in practice, in view of the strategies we presented in Chapter 2. The research questions guiding our investigation are:

PQ1 Do large real-world projects have a convention about the order of methods inside classes?
PQ2 If they do, which conventions are common?

To answer these questions, we define a normalized scoring function that quantifies, given a class and a strategy, how well does the order of methods in the class fit the strategy. We then report on our findings where we applied the scores to a large body of code.

3.1 Scoring Functions

Definition 2 (Normalized scoring function). Given an ordering strategy $s$ and a class $C$, a normalized scoring function returns a score, representing how well is class $C$ ordered by the strategy $s$. The score is normalized to a number between 0 (the class is not ordered at all by the given strategy) and 1 (the class is fully ordered by the given strategy).

To compute the score, for each strategy, we compute the distance in terms of inversion counts between the order of methods in the given class and the
order of the same methods according to the strategy. Specifically, we count how many of the pairs of methods that should be ordered by the strategy are not in the correct order in the given class. To normalize, we divide this inversion count by the total number of possible method pairs. Formally:

$$\text{Score} = 1 - \frac{\text{wrongOrdPairs}}{\binom{\text{methodsCount}}{2}}$$

(3.1)

where for strategy $s$, $\text{wrongOrdPairs}$ is the number of method (distinct) pairs whose order is wrong, i.e.,

$$\text{wrongOrdPairs} = |\{(i, j)|i < j \land m_i >_s m_j\}|$$

An alternative method to compute scores using inversion counts, would be to separately count and remove all pairs whose order is undefined. Formally:

$$\text{Score} = 1 - \frac{\text{wrongOrdPairs}}{\binom{\text{methodsCount}}{2}} - \text{undefinedPairs}$$

(3.2)

where for strategy $s$, $\text{wrongOrdPairs}$ is the number of method (distinct) pairs whose order is wrong, and $\text{undefinedPairs}$ is the number of pairs whose order does not matter.

We chose not to use this alternative scoring function because it is not monotonous: adding more constraints to the ordering strategy does not necessarily mean we would get an equal or lower score. For example, with this scoring function, SC can have a higher score than SC-R, even though the former has more constraints than the latter.

Below we give examples for scoring using the class shown in List 2.1 and the four strategies defined in Chapter 2. We denote the class’ methods $m_1$ to $m_7$, according to their order of appearance in List 2.1.

StyleCop (SC). The StyleCop strategy defines a full order on the 7 methods in this class, therefore the scores’ denominator is $\binom{7}{2} = 21$. In List 2.1 exactly 2 pairs of methods are presented in a wrong order according to the strategy: $(\text{doThis}, \text{doThat})$ and $(\text{doNothing}, \text{doThat})$. Both are wrong because static methods should come before non-static methods (within the same group of access modifier). Thus the resulting score is: $1 - \frac{2}{21} = 0.9041$.

Note that the same score applies to the application of the Relaxed-StyleCop (SC-R) ordering strategy, since the 2 wrongly ordered pairs are also wrongly ordered according to SC-R.
3.1. SCORING FUNCTIONS

Calling (CL). We build the directed call-graph for the class. To enforce that in this strategy constructors will come first, we add edges from methods \( m_1 \) and \( m_2 \) to all other methods. The order between any two methods that are not connected on the graph or two methods that are on a cycle, is undefined. The order between any two methods that are connected on the graph but are not on a cycle is defined according to the direction of the path between them.

In our example, we have an edge between \( m_1 \) and \( m_2 \) (the constructors) to all the others, an edge \( \langle m_4, m_3 \rangle \), and an edge \( \langle m_5, m_7 \rangle \). The number of wrongly ordered pairs is 1: only \( \langle m_4, m_3 \rangle \) is in an inverted order (the methods realExampleMethod and exampleMethod). Thus, the resulting score is \( 1 - \frac{1}{21} = 0.9523 \).

Connectivity (CT). We denote the 4 connected components of the undirected call-graph of the class by \( c_1 \) to \( c_4 \), such that \( c_1 = \{ m_1, m_2 \} \), \( c_2 = \{ m_3, m_4 \} \), \( c_3 = \{ m_5, m_7 \} \), and \( c_4 = \{ m_6 \} \). Recall that the order within connected components and between connected components is not defined by the strategy. Rather, the requirement is that methods in the same component will be grouped together. Thus, we are left with only 3 pairs of methods whose order is defined by the strategy: \( \langle m_1, m_2 \rangle, \langle m_3, m_4 \rangle, \langle m_5, m_7 \rangle \). We denote the connected component of method \( m_i \) by \( CC(m_i) \).

We can see that \( CC(m_1) = CC(m_2) \neq CC(m_3) = CC(m_4) \neq CC(m_5) = CC(m_7) \neq CC(m_6) \). For a pair \( \langle m_i, m_j \rangle \) s.t. \( 1 \leq i < j \leq 7 \), the order is relevant only if \( CC(m_i) = CC(m_j) \), and the order is wrong if there is a \( k \) s.t. \( i < k < j \) and \( CC(m_i) \neq CC(m_k) \).

Thus, \( wrongOrdPairs = |\{ \langle m_i, m_j \rangle | i < j \wedge CC(m_i) = CC(m_j) \wedge (\exists k. i < k < j \wedge CC(m_i) \neq CC(m_k)) \}| \). In this case, the pair \( \langle m_5, m_7 \rangle \) is in a wrong order, because \( CC(m_5) = CC(m_7) \) and there is a \( k = 6 \) s.t. \( CC(m_5) \neq CC(m_6) \). This is the only pair with a wrong order, so the resulting score is \( 1 - \frac{1}{21} = 0.9523 \).

Calling+Connectivity (CLCT). This strategy combines CL and CT. We build both the directed call-graph (for CL) and the undirected call-graph (for CT). When we compare methods \( m_i \) and \( m_j \) we first see the results for each strategy CL and CT separately. If both suggest the order is undefined, then it stays undefined. If one of them suggests the order is wrong, then the order between \( m_i \) and \( m_j \) in this strategy is also wrong. In any other case, the order of the methods in this strategy is the result defined by CL or CT.

In our example, we have 13 relevant method pairs for this strategy. CL has 12 relevant pairs, CT has 3 relevant pairs, but only 1 out of the 3 pairs (
\(\langle m_1, m_2 \rangle\) is a pair that was not relevant at \(CL\). So in total we have 13 relevant pairs. We count the same wrong orders from before: \(CL\) had one wrong order (\(\langle m_3, m_4 \rangle\)), and \(CT\) had one wrong order (\(\langle m_5, m_7 \rangle\)). Together we have 2 pairs with a wrong order. Thus, the resulting score is: \(1 - \frac{2}{21} = 0.9047\).

### 3.2 Empirical Study

**Corpus.** For our study we used the Qualitas Corpus [27], available from [22]. The corpus contains 112 open-source projects such as `ant`, `jboss`, `jfreechart`, `jhotdraw`, `junit`, `log4j`, `hadoop`, `hibernate`, `tomcat`, `weka`, `xerces`, etc. comprising more than 100,000 Java classes. Specifically, we used version QualitasCorpus-20130901r.

**Execution.** To compute the score for each class we extract relevant information from its source code (method names and arguments, modifiers, line numbers, etc.). We build a call-graph for the class based on the code’s AST. Then, for each strategy, we compute the score according to Equation 3.1 defined above. When analyzing each class, we differentiated between methods by their signatures (name and number of arguments), and if more than one method conformed to the same signature (for example, the only difference is the first argument’s type), we treated same-signature methods differently: the pair was wrongly ordered if there was a different method signature between them in the class. This is because we believe these methods should be treated as the same, thus they should be grouped together (as in the CT strategy).

### 3.3 Results and Observations

**Corpus statistics.** The corpus has 128,818 Java files in it. We discarded all 28,218 classes with less than 2 methods in them as method ordering is irrelevant for these classes. We further discarded all enums and interfaces. 15,919 files had more than one class in them, and we considered only the first class they included (in particular, ignoring inner and anonymous classes). Thus, we used only 100,600 files of relevant classes.

For the relevant classes, there was an average of 10.48 methods per class, and a median of 6 methods per class. In addition, among the relevant classes, there was an average of 30.5 method calls (and median of 11) per class. Also
3.3. RESULTS AND OBSERVATIONS

<table>
<thead>
<tr>
<th></th>
<th>SC</th>
<th>SC-R</th>
<th>CL</th>
<th>CT</th>
<th>CLCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.70</td>
<td>0.91</td>
<td>0.96</td>
<td>0.95</td>
<td>0.92</td>
</tr>
<tr>
<td>Weighted average</td>
<td>0.62</td>
<td>0.91</td>
<td>0.94</td>
<td>0.85</td>
<td>0.80</td>
</tr>
<tr>
<td>Median</td>
<td>0.70</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>STD</td>
<td>0.24</td>
<td>0.16</td>
<td>0.11</td>
<td>0.09</td>
<td>0.14</td>
</tr>
<tr>
<td>% of &gt; 0.95 (all)</td>
<td>23.3%</td>
<td>64.4%</td>
<td>79.9%</td>
<td>77.3%</td>
<td>66.6%</td>
</tr>
<tr>
<td>% of &gt; 0.95 (10-)</td>
<td>31.9%</td>
<td>72.4%</td>
<td>82.6%</td>
<td>87.0%</td>
<td>75.6%</td>
</tr>
<tr>
<td>% of &gt; 0.95 (11+)</td>
<td>2.0%</td>
<td>44.4%</td>
<td>73.3%</td>
<td>52.9%</td>
<td>43.8%</td>
</tr>
</tbody>
</table>

Table 3.1: Scores for the five method ordering strategies, applied to the 100,600 relevant classes from the Qualitas Corpus [27]

an average of 6.4 local (inside the class) method calls per class, and an average of 3.4 distinct local method calls per class.

Results. Table 3.1 shows the results of computing the score over all the relevant classes in the corpus, for the strategies defined in Chapter 2, using the scoring functions defined in 3.1. For each of the strategies, we show the average, weighted average, median, and standard deviation of the scores, as well as the percentages of classes that received scores above 0.95 (in total, for classes with 2-10 methods, and for classes with 11 and more methods). Weighted average computed using weights based on number of method-pairs considered.

In total (not shown in the table), 93.2% (97.6%) of the relevant classes in the corpus have received a score greater than 0.95 (0.90) for at least one of the strategies.

Observations. Based on the above results, we are able to provide partial answers to our questions.

The answer to PQ1 is positive. The results show that in more than half of the classes, the order of methods gets a perfect score in at least one of the strategies we considered. Moreover, almost all classes conform almost perfectly to at least one of our strategies.
To answer PQ2, the results show that the CL ordering strategy has a higher score than the other strategies, with an average of 0.96 (weighted average 0.94), and a rather small standard deviation of 0.11. Furthermore, for CL, more than 79.9% of classes have received a score greater than 0.95. This provides evidence that the CL ordering strategy is relatively common in real-world software projects, while the other strategies we considered are less common.

3.4 Threats to Validity

We now discuss threats to the validity of our results, divided into construct, internal, and external threats [14].

Construct threats refer to the degree to which the operationalization of the measures actually represents the constructs in the real world. Internal threats refer to the extent to which the treatment or independent variables were actually responsible for the effects seen (unknown factors may have had an influence on the results). External threats refer to the degree to which the findings can be generalized to other populations or settings (rather than the controlled environment).

Construct. First, we might have missed some ordering strategies which could have shown us more interesting results. To address this threat, we searched for suggestions from known tools and literature and attempted to formalize what we have found. Second, different scoring functions might have yielded different results. As we explain above, we chose a monotonous scoring function (more constraints means lower scores) that takes into account all method pairs and not only ones whose order is constrained. Third, the algorithms we designed might not fully reflect each strategy’s real intention. This applies in particular to the CT strategy and the notion of related methods (see future work mention of feature extraction). Fourth, some Java IDEs allow the engineer to create a new method by writing a method invocation of a non-yet-existent method signature; the tool creates a default implementation of the new method right after the calling method. This means that our results in favor of the CL ordering strategy might represent the extent use of specific tools rather than a convention projects or engineers follow intentionally.
3.4. THREATS TO VALIDITY

**Internal.** First, our implementation of computing scores might have bugs. To mitigate this, we used JUnit [16] to test the main methods and different ordering algorithms. We wrote 6 Java classes: one that would get 0 in all scores, one that would get 1 in all scores, two with specific edge cases and two with different scores focusing on the different variables concerning each strategy. We manually computed each strategy’s score for these classes and compared it to the output from the implementation. Second, our computation of call-graph is light-weight, and does not consider features such as overloading and overriding. All methods with the same name and number of arguments are represented as one in regards to method calls (whether they are the caller or the callee), but when computing scores we compare them individually. We believe methods with same name and number of arguments should be treated as one. When overloading with different number of arguments the order may be important (e.g., methods with less arguments call methods with more). Third, it may be that some of the classes we analyzed contain unreachable code or do not compile within their projects. This may have affected the reliability of our results. Fourth, for simplicity we supported analyzing only one class per file. We only considered the first, ignoring any additional classes (including inner or anonymous classes).

**External.** First, we only considered and analyzed Java code. Ordering conventions may be different in other languages. Second, we exclusively analyzed open source projects. Other projects might have different coding styles and orders used and enforced.
CHAPTER 3. METHOD ORDERING STRATEGIES IN PRACTICE
Chapter 4

User Study

The research questions guiding our user study are:

RQ1 Are some orders better than others in improving code comprehension correctness?

RQ2 Are some orders better than others in reducing the time spent on comprehension?

RQ3 Does engineers’ experience affect their sensitivity to ordering?

To answer these questions we conducted the following experiment.

4.1 Experiment Setup and Execution

In the experiment setup and execution described below we tried to avoid some of the confounding parameters recently surveyed in [24].

Experiment setup. We selected 5 classes from 5 different open-source projects, as listed in Table 4.1. The number of methods per class ranged from 5 to 33 (average 22.4). The number of lines of code, excluding comments, which we have removed from the code, ranged from 142 to 579 (average 312.8).

In addition to the original source code order of each class, we created 4 more variants of each class, each using one of the ordering methods: SC, CL, CLCT, and Random (a strategy which assigns a random method order).

We further prepared 3 multiple-choice comprehension questions for each of the 5 classes. Each question had a correct answer, a partly correct answer (almost correct), a wrong answer, and a fourth option of “I don’t know”.

17
Table 4.1: Classes used in user study, with number of methods #M and number of lines of code #LOC

<table>
<thead>
<tr>
<th>Project</th>
<th>Class</th>
<th>#M / #LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeAnalyzer</td>
<td>repository.Cache</td>
<td>19 / 320</td>
</tr>
<tr>
<td>Checkstyle</td>
<td>com.puppycrawl.tools.checkstyle.checks.ClassResolver</td>
<td>5 / 142</td>
</tr>
<tr>
<td>Servlets</td>
<td>com.oreilly.servlet.MailMessage</td>
<td>27 / 238</td>
</tr>
<tr>
<td>Azureus</td>
<td>org.gudy.azureus2.ui.swt.progress.ProgressReporter</td>
<td>33 / 579</td>
</tr>
<tr>
<td>Eclipse</td>
<td>org.eclipse.core.internal.localstore.UnifiedTree</td>
<td>28 / 285</td>
</tr>
</tbody>
</table>

The first and second questions for each class were about its functionality. The third was related to the addition of new functionality.

To avoid the case where the questions are too easy for participants, to the extent that all are answered correctly and fast regardless of the order of methods in the presented code, we designed the questions to be rather difficult. We have shown drafts of our questions to several colleagues to assess their difficulty before we finalized and executed the experiment. All our colleagues considered the questions to be difficult. Some questions were considered too difficult or took more than 5 minutes to answer. We revised our questions based on feedback from our colleagues.

To illustrate the questions’ difficulty level, we show here the text of three questions.

1. Say someone called setMaximum(100) and then setPercentage(200, “Cheat”). What would happen to the reporterListeners (in regards to add\remove from that list), which would report (via the report method) that they are OK to dispose? And how would the messageHistory change given its current size is 1000?

2. Given a ProgressReporter instance named pr, give an example for two consecutive public method calls: One which notifies listeners and another set method (besides setReporterType) which would not notify listeners. Hint: Second method call can be a result of the first method call. Find when updateAndNotify(...) does not notify listeners.

3. Change the method addNodeChildrenToQueue(...) by filling the missing lines. The new code should implement the following feature: After adding a node’s children to the queue, use childrenMarker to test if the node is the last of its brothers in queue, and if so - remove the marker from queue. In this case you should add a levelMarker to queue if the next element is a level marker.
The complete set of classes used (in all orders) and the text and answers of the questions we asked is available in [http://smlab.cs.tau.ac.il/ordi/](http://smlab.cs.tau.ac.il/ordi/).

Every participant was able to see each class presented using an ordering strategy selected in random. After answering questions about 2 classes, the participant could continue answering up to all 5 classes. The order of classes presented was also random in order to get enough answers per class (in case of drop-outs). For each presented class, in its specific method ordering, the participant was asked 3 questions. We asked the same 3 questions in the same order for each class, regardless of the presented method order.

We implemented the questionnaire in a web-based split-screen interface, showing the color-coded class code on the left (allowing searching and scrolling), and one question at a time on the right.

Finally, in addition to recording the participant’s answers to the questions, we have also recorded the time spent on answering each question.

**Experiment execution and participants.** We have put the web-based questionnaire online and published it in mailing lists of software engineers in industry. Participation was anonymous and offered no reward. We did not tell the participants what we are trying to evaluate, except that we asked them to do their best in answering the comprehension questions.

Overall, within 3 weeks, we have received answers from 60 engineers, none of them students in our university. The 60 engineers answered a total of 328 questions, where the average time spent per question was 3.02 minutes (0.87, 2.15, and 4.12 minutes for the 25th, 50th, and 75th percentiles resp.). On average, we got 4.47 answers per order per class, too few samples to consider statistical significance.

The 60 participants who answered the questionnaire came from 12 countries, 35% from Israel and the others from Germany, India, Ukraine and more. 13% of the participants were females, one chose not to specify a gender. In terms of experience in Java, 3 subjects (5%) defined themselves as having no experience, 35% defined themselves as beginners, 51% as professionals, and 9% as experts.

### 4.2 Results and Conclusions

**RQ1: Effect on correctness**

Figure 4.1 reports the correctness results for the first question over all classes, summing up 117 answers in total. The chart shows an advantage
CHAPTER 4. USER STUDY

Figure 4.1: Correctness for the 1st question, showing the percentages of correct, partly correct, incorrect, and ‘don’t know’ answers, for 5 method orders

![Correctness Chart](image)

Figure 4.1: Correctness for the 1st question, showing the percentages of correct, partly correct, incorrect, and ‘don’t know’ answers, for 5 method orders.

to the CLCT order compared to the other orders. For the first question we asked on each class, CLCT had the least percentage of wrong answers and the least percentage of do not know answers; it had over 80% correct or partly correct answers, compared to around 60% for all other orders.

Based on this data, we consider the results to be inconclusive. It seems that for the first question on each class, CLCT order resulted in better results in terms of correctness. However, the results for the second and third question do not show a similar phenomenon. One way to interpret these results is to consider a learning effect: the better order affects the first-time reading of the code; After the first reading, the engineer already knows the code to the extent that the quality of subsequent readings is no longer affected by the order of methods, for better or worse.
To answer RQ1, the results do not show correlation between method ordering and correctness of comprehension. For the first question asked on each class, results for the CLCT order are best in regards to correct or partly correct answers. However, this is no longer the case for the second and third questions, perhaps due to a learning effect.

RQ2: Effect on time spent

Figure 4.3 shows normalized times for the 1st (top), 2nd (middle), and 3rd (bottom) questions, comparing five orders, SC, Random, Original, CLCT, and CL, across all classes. Each box plot shows the 5th, 25th, 50th (median), 75th, and 95th percentiles. Note that we normalize the times for every class/question pair separately. Then, we aggregate per question (first, second, and third) and for each, split into a box plot per method order. For example, if we got the following times (in seconds) for the first question of class $C_1$, say $q_{1,1}$: 10, 23, 25, 30, and the following times for the first question of class $C_2$, say $q_{2,1}$: 13, 22, 33, 45, then for question $q_{1,1}$ we will get the normalized times of 0.0, 0.65, 0.75, 1.0, and for question $q_{2,1}$ we will get the normalized times of 0.0, 0.28125, 0.625, 1.0. Both are for the first question so they will be counted for the first set of box plots (at the top of Figure 4.3) however distributed between the five box plots based on the method order they
CHAPTER 4. USER STUDY

Figure 4.3: Normalized times per answer for the 1st (top), 2nd (middle), and 3rd (bottom) questions, comparing five orders across all classes. Each box plot shows the 5th, 25th, 50th (median), 75th, and 95th percentiles.

Based on this data, we observe that for all three questions, the CL and CLCT orders provide minimal times at the 25th percentile. We also see that the median for CL and CLCT is always smaller than the median for the original order of methods in the class.

Further, when comparing CL and CLCT, the 25th percentile for CL is always smaller than the 25th percentile for CLCT, while the 75th percentile and the 95th percentile for CL are always greater than the 75th percentile and the 95th percentile for CLCT. Thus, CLCT provides lower variance than CL.

Another observation is that the Random order results in worst minimal times, at the 5th and 25th percentiles, compared to all other orders, across
4.2. RESULTS AND CONCLUSIONS

all three questions.

As in the discussion above about correctness, there may be a learning effect that blurs the differences between orders in the second and third questions asked about the same code. Thus, the results for the first question should be considered the most valuable in this regard.

To answer RQ2, the results seem to show that the order of methods affect the time spent on comprehension. Compared to all other orders, CLCT and CL orders allow many engineers to reduce the time required for comprehension at the 25th percentile. A random order results in worst 5th and 25th percentile times.

RQ3: Sensitivity to experience

As mentioned earlier, 40% of the 60 study participants have stated that they have almost no experience or only beginners experience in Java, while the remaining 60% have defined themselves as professional or experts. Below we name the first group “beginners” and the second group “experts”. We set out to examine whether engineers’ experience affect their sensitivity to method order.

First, as a self validation check, we report that in total, beginners were indeed less likely than experts to be correct or partly correct in their answers. Specifically, while only 55% of beginners’ answers were correct or partly correct, 66% of experts’ answers were correct or partly correct. This gives us some confidence in the validity of the results below.

Second, while we did not observe interesting results in terms of the effect of experience on correctness per method order (and so we do not show these results here), we did observe what seems to be a major difference between beginners and experts in terms of the effect of method orders on the time spent on answering.

Specifically, Figure 4.4 shows normalized times for the 1st question for beginners (top) and experts (bottom), comparing five orders, SC, Random, Original, CLCT, and CL, across all classes. Each box plot shows the 5th, 25th, 50th (median), 75th, and 95th percentiles. The box plots summarize the results from 50 answers by beginners and 67 answers by experts. We first separated the times for beginners and for experts, then normalized the times as before in Figure 4.3. Thus, we normalized separately for each of the two groups of participants.
Based on this data, looking at the lower 5th and 25th percentiles, as well as at the higher 75th and 95th percentiles, we observe that for beginners, the method orders of SC, CLCT, and CL, show reduced time spent on answering, while for experts, we do not observe such effect. Complementarily, beginners suffered most from the Random and Original orders, while for experts, these two orders seem not significantly different than the other three orders. As a more formal measure of the difference, the variance for the values between the 25th and 75th percentiles for beginners across the five method orders is 0.04, while for experts it is 0.02.

To answer RQ3, the results show that more experience implies less sensitivity to method order in terms of time spent on answering comprehension questions. It also shows that the SC, CLCT, and CL orders help beginners in spending less time on answering, while the Random order causes beginners to spend more time on answering.

### 4.3 Threats to Validity

We now discuss threats to validity of our answers to the research questions and additional limitations of our work, divided, as in Section 3.4, between
4.3. THREATS TO VALIDITY

construct, internal, and external validity threats.

**Construct.** First, our time measurements might not be reliable. To mitigate this in advance, we created a Pause\Continue button, which participants could use. Still the correct use of this button depends on the participant. Second, the different classes and the different questions per class might have high variance in level of difficulty. This may have created biases in the comparisons of answers and times. To mitigate this, we normalized time measurements per question per class as explained above. Finally, our automated reordering tool removes comments from the class source code. Comments may affect code comprehension in different classes in different ways. We partly addressed this threat by removing the comments from the original version of each class as well.

**Internal.** First, we used multiple-choice questions, which allow participants to guess an answer even if they do not know it. To partly address this, all answers included an “I don’t know” option. Still multiple-choice questions may not be representative of comprehension tasks in practice. Moreover, in terms of timing, some participants might get tired or uninterested after a few questions and so they may choose a random answer or “I don’t know” quickly, just in order to advance and finish. Second, we used one random method ordering strategy and three method orders we have defined in this Thesis. Other method orders could have given us different results. We partly addressed this by examining the use of these orders in practice and finding that they are common. This motivates the use of these orders and not others in the study.

**External.** First, the questionnaire does not fully reflect a real-world setup, as it was done on a web browser. Real-world projects have rich IDEs to help navigation (to definitions or invocations). In many cases also an outline of methods is available. Our present work eliminates the IDE factor. We kept the environment to a minimum: color coding and simple text search. The advantage is that the results do not depend on any specific IDE. The disadvantage is that the setup does not fully reflect a real-world one. Second, the questionnaire uses only Java as a programming language. One may receive different results when experimenting with scripting or other languages. This limits the generalizability of our results. Still Java is a most popular language so we consider the results to be useful.
CHAPTER 4. USER STUDY
Chapter 5

Prototype Tool: Ordi

Tool description. We have implemented our work in an Eclipse plugin named Ordi. Ordi provides two main functions. First, an engineer can select a resource (Java project, package, or class file) and a method ordering strategy and ask Ordi to check whether (and to what extent) does the selected resource conform to the selected strategy. Ordi will compute and output the relevant normalized score. Second, an engineer can select a Java class file and a strategy and ask Ordi to reorder the methods in the class according to the selected strategy. Both functions are integrated into Eclipse’s standard Refactor menu.

As an example, one can select a class, right click to open the context menu and select Refactor/Method Ordering, then select Reorder and the Calling strategy. Ordi will reorder the selected class according to the strategy. Then, if she tests that class to see to what extent it is ordered by the Calling strategy, she will get a score of 1.0.

In addition to individual, local refactorings, available to engineers, Ordi can be used by team leaders or development managers to define and enforce a consistent ordering strategy, as part of a more general project-wide code convention.


Limitations. Ordi is a prototype Eclipse plugin whose main purpose is to present and test our work. The current implementation has the following limitations: it supports the Java language only and performs on exactly one
class at a time; it assumes that `.java` files contain only one class; it does not support special annotation syntax (with @); it ignores enums and interfaces; and it removes comments, as they are not part of the AST that it builds (to partly compensate for that, before changing a file it saves a backup with its comments). Further implementation to overcome these limitations is outside the scope of this Thesis.

**Availability.** Ordi is compliant with the latest Eclipse version 4.5.1. We made it available for download from [http://smlab.cs.tau.ac.il/ordi/](http://smlab.cs.tau.ac.il/ordi/). We encourage the interested reader to try it out.
Chapter 6

Related Work

We discuss existing literature on ordering and on other topics related to program comprehension and how it is measured.

6.1 Ordering

Biegel et al. [4] examine different ordering strategies of methods and fields inside Java classes. They suggest several predefined ordering criteria, including JCC [20] (Java Coding Conventions - by entity type), Visibility, Lexicographic, Method Subcategories (types of methods), and Semantic Similarity (clustering by TF-IDF).

The JCC criteria is considered a primary criteria and it distinguishes between 4 categories: class variables (static attributes declared at class level), instance variables (non-static attributes declared at class level), constructors (distinguished methods to create instances of a class), and all other methods.

List 6.1 shows an example class with entities ordered by the JCC criteria of Biegel et al.

The Visibility criteria is secondary (meaning that it is used after ordering by the JCC). It suggests that class and instance variables should be ordered by visibility: public, protected, default, and private (much like the SC strategy that we have defined and used). The Lexicographic criteria is also secondary. It suggests that each group’s entities should be sorted alphabetically. The Method Subcategories criteria is another secondary criteria. It suggests that methods should be sorted by the following categories: static method, initializer, getter/setter, and other methods. Finally, the Semantic Similarity is
another secondary criteria for methods. It suggests that the methods should be grouped by their functionality. For this, Biegel et al. use TF-IDF (term frequency-inverse document frequency), which is a widely applied information retrieval technique. TF-IDF balances the importance of each word so that words occurring in many methods become less relevant. By using TF-IDF and a clustering algorithm in which each method is a vector of words occurrences, they split the methods to four groups, such that inside each group there are methods with similar identifiers.

Our work is very different yet may be viewed as complementary to \[4\] in some ways. We focus only on methods. Our interpretation of the ordering conventions is different and is largely based on the control-flow graph of the methods in the class. In their work, they interpreted the vague JCC sentence that methods “should be grouped by functionality rather than by scope or accessibility” as similar functionality (the Semantic criterion which they measured by TF-IDF), while we interpret it as related functionality, based on connectivity in the control-flow graph (the CT strategy).

The way Biegel et al. assess how much a class is ordered by a strategy is different than ours. In this respect, our ordering strategy is similar to their ordering criteria. When assessing a specific criteria, they gave each entity a color (one out of 3-4 categories) that defined its group. While viewing the result graphs, it is possible to see if a class is properly ordered, as colors should be grouped separately in a specific predefined order. Later, they defined a Sort Metric to automatically give a score to each criteria in each
class. With this metric they compared only adjunct entity pairs, comparing the entities’ group index. In this Thesis, we provide a formal scoring function giving a normalized score (a number between 0 and 1), taking into account all method pairs: we compare each pair separately. Each method pair could be compared by a lot of factors and is not necessarily limited to only a few groups. Thus, our scoring function may be viewed as global while theirs is local, as ours has a somewhat more refined granularity.

Finally, Biegel et al. suggest that “The next step would be to better support developers in applying a certain ordering strategy, for instance, by providing semiautomatic ordering tools”. Indeed we provide such a tool.

6.2 Identifier Names, Length, and Style

Lawrie et al. [17] study the effect of word-based identifiers (full-words, abbreviations, single letters) on comprehension, the effect of gender on comprehension and confidence, and the effect of experience on sensitivity to identifiers. They study these via an experiment with human subjects, each trying to describe code with different identifier variants. Their study suggests that men perform significantly better on full-words identifiers (while with women it is not statistically significant), that women rate their confidence lower than men, and that experience plays part in confidence ratings and not in correctness.

Lawrie et al. measured comprehension by asking subjects to describe a given code. The experiment showed each subject an algorithm (method) or a code snippet, and then asked them for its description and confidence level on their answer. In addition to a rating on the subject’s answer, they also kept track on experience, gender, and confidence of each participant. Their experiment is similar to our experiment in that it aimed at testing comprehension by showing code and asking a comprehension question about it, while recording correctness and additional information on the participant. It is also very different from our experiment, as they did not measure the time spent on answering, they did measure confidence level, and the answers to their questions were open (while we used closed multiple-choice questions). They also tested for significance with statistical models, which we did not do.

Binkley et al. [7] studied the effect of code line length on memory accuracy and time. They also tested a connection between experience and “memory
ties” to these variables. They study these effects via an experiment, testing subjects’ memory of a line of code, in correlation to the line’s length. Their results suggest that the number of syllables affect comprehension time, and to a lesser extent the correctness of recalling an identifier. They also found that “ties” and experience improve recall correctness, and that as the length increases so does the influence of experience.

Binkley et al. measured comprehension by testing for correct and fast memory of identifiers in lines of code. The experiment showed subjects a long line of code, and then after a distraction (writing a paragraph), asked them to complete the line with an identifier. Both correctness and time were taken into account. The experiment was similar to ours in the parameters it recorded (correctness, time, experience) and the way it recorded it (split screens). But it was different in the way of statistical analysis, and how it tested “comprehension”, by testing one’s memory and not by asking comprehension questions.

In another work by the same group, Binkley et al. [5] studied the impact of identifier style on effort and comprehension. They used different methods and techniques, like SAT passage comprehension, method’s summarization, identifier’s identification, identifier memorization, and the use of eye-tracking. After a number of studies, the accumulated evidence suggested that camel-Case style is better for comprehension than underscore style, especially for beginner programmers.

In this study, comprehension was measured in different ways, including the understanding of a passage in a natural language, summarizing a method correctly, recognizing identifiers in clouds or lines of code, recording eye fixations and testing lines memorization. In this paper they report on 5 different studies, each one had an experiment of its own, all had human subjects tested. Two experiments included [7] and [23], showing a phrase and asking the subject to find it in a moving cloud with a specific style. Another experiment was a passage with SAT questions, when some phrases were changed according to a specific style. Another experiment asked to find identifiers in a lot of lines of code, and the last experiment asked to summarize methods. All tested for time and correctness, some checked for correlation to experience.

These experiments are different than our experiment in a few ways. First are the statistical models built for significance testing. Second, some tested for comprehension in different ways (e.g., readability, eye tracking, and passage understanding in a natural language rather than code), while we used comprehension questions about real code. The experiments were similar to
6.3. LINGUISTIC ANTIPATTERNS

Ours in that all measured correctness and times, and some measured, like us, a correlation to experience. Another similarity is in some of the experiment designs; using human subjects to answer online questions, using a kind of questionnaire with multiple screens.

Finally, Binkley et al. study the effect of identifier styles (camelCase, under_score) on time and correctness of finding phrases. They also study the effect of training (experience) on the effectiveness of style on these parameters. They test it via an experiment, showing a phrase to each subject, and then with floating clouds of phrases in a specific style, one needs to select the same phrase (correctly and quickly). They suggest that camelCase style leads to better all-around performance once a subject is trained on this style.

Binkley et al. did not measure “comprehension” per say; instead they measured “developers effectiveness” in finding phrases correctly and quickly. They tested both accuracy (correctness) and time and whether the experience level affects them. The experiment design was: given a phrase, find it in camel-casing (distractors with low edit distance) or in under_score, in a randomly floating cloud. Then they used statistical models (GLMM) and tested for significance of the null hypothesis. It is different from what we did in that the experiment was more focused (a lot more isolated) and they used statistical models to test significance. It was similar to ours in that they tested correctness and times in addition to the effect of experience, and they had an online testing environment like ours, with several screens.

6.3 Linguistic Antipatterns

Arnaoudova et al. introduces the concept of Linguistic Antipatterns (LAs). Linguistic Antipatterns in software systems are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of an entity, thus possibly impairing program understanding. They detail one family of LAs intended as recurring inconsistencies in method name, signature, and comments, as well as in attribute name, signature, and comments. For methods, it is categorized to cases where a method (i) does more than it says, (ii) says more than it does, and (iii) does the opposite than what it says it does. For attributes, it is categorized to cases where an attribute (i) contains more than it says, (ii) says more than it contains, and (iii) contains the opposite of what it says it contains.

They suggest testing code for possible LAs, and for that purpose they
implemented a tool called LAPD, to analyze java source code. They studied to what extent 3 analyzed systems contain the LAs they defined, with a precision of 72% (the implementation is challenging and questionable as it is difficult to capture opposing meanings). Their study is mainly similar to our preliminary research, where we defined ordering strategies and analyzed whether and how does the code in an existing corpus follow the defined orderings as a convention. One may view bad ordering as a kind of a linguistic antipattern.

In recent work by the same group, Arnaoudova et al. 2 studied how developers perceive LAs. They conducted two experiments investigating whether developers perceive LAs as poor practices, and if this is the case, whether they would take any action to remove LAs. The experiments were online questionnaires showing code snippets and asking questions that determine whether the LA (if any) was observed, what the subject’s opinion on it was, and if the subject would modify it. They suggest that the majority of the subjects (69% in the first study, 51% in the second study) indicated that the provided LAs are poor or very poor practices. They also found three LAs most developers agree on and perceive as particularly poor - LAs concerning the state of an entity (attributes) that belong to the categories “says more than what it does” and “contains the opposite”. The studies did not evaluate code comprehension; rather they relied on developers’ opinions. Both studies were given to participants as an online questionnaire about open source code snippets. While the first study had external participants (i.e., participants not familiar with the code), the second study had only internal participants (i.e., people developing or maintaining the code). In the studies, participants were shown several code snippets and were asked questions on each. Most of the snippets had LAs in them. The first question was about the subject’s opinion on the LA (without being aware of what an LA is), and there were follow-up questions to verify their meaning and if and why they would take action to modify it.

These experiments were similar to ours mainly in how they were done: an online questionnaire, showing random ordered source code and asking questions about it. When analyzing the results they examined the effect of experience. On the other hand, in their experiments they supported not only closed but also open questions, and when they analyzed the results they examined the effect of subject’s main programming language and their occupation, which we did not examine. In addition, in our research we used the questions to evaluate subjects’ program comprehension, while their experi-
ments mainly asked for the subjects’ opinions.

6.4 Eye Tracking

A working session from ICPC 2009 [10] introduced the use of eye tracking for code comprehension studies. It suggested that since eye-tracing tools are of high quality, accurate, and user friendly, they can be used to understand the cognitive process involved in the processing of visual data. It can be used to capture fixations (stabilization of eyes on an object of interest for a period of time), saccades (quick movements of the eyes from one location to the next), and scan-paths (directed path formed by saccades between fixations). The general consensus in the eye-tracking research community is that the processing of visualized information occurs during fixations, whereas no such processing occurs during saccades.

Busjahn et al. [8] study the linearity of natural language readings compared to source code readings. Linearity represents how closely readers follow a text’s natural reading order (top-to-bottom, left-to-right path). They use eye-tracking tools for this study, recording multiple measurements on eye fixations, saccades, and scan-paths while subjects read and try to comprehend text. They found that 80% of the novices’ eye movements were linear when reading natural language text and 70% when reading source code. On the other hand, for experts, they found only 60% linear eye movements on source code. They suggest the experts’ reading patterns can be characterized by a greater number of eye gaze movements that skip intermediate words and lines.

In their experiment, they gathered both novice and experienced programmers as subjects, and handed each several English passages and programs. After each English passage there was a single comprehension question, and after each program was a code comprehension question (summarization, the value of a variable after execution, or a multiple-choice question about the algorithmic idea). The order of the readings was random. Amongst the parameters they recorded per each text were Element Coverage (percentage of words covered in passage by subject) and Saccade Length (average distance between two eye fixations). Then, they measured “How much the scan-path followed Story Order of Execution Order” using the matching algorithm called Needleman-Wunsch, where Story Order is the basic top-down left-right path and Execution Order is the control-flow of the source code.
Similar to their experiment, our experiment tested for the effect of programming experience on the results. In addition, both showed source codes and asked comprehension questions about them, only in their experiment the answers matter less (as they recorded the eye-tracking and cared less about the correctness of comprehension and more about the reading process). We measured time while they did not measure time. Also, the idea behind the comprehension questions was similar.

Our CLCT strategy may be able to minimize scrolling and eye movements. Thus, investigating the relationship between method order and eye movement may be an interesting direction for future work.

6.5 Code Regularity

Jbara and Feitelson [12] introduce a new code complexity metric named code regularity, and study its effect on program comprehension. Code regularity is a measure for quantifying when the same structures are repeated in a code (they quantified it by using a compression ratio). Similar to LOC (lines of code) and MCC (McCabe Cyclomatic Complexity) it is a static program metric that might affect its complexity. They study the questions of how this metric affects code comprehension (in terms of correctness and time) and what is its effect relatively to other code complexity metrics (can regularity compensate for other measures like LOC and MCC?). They created a controlled experiment, testing for code comprehension amongst human subjects, while different source code shown contained different measures of regularity, LOC, and MCC.

Jbara and Feitelson suggest that high MCC and LOC do not fully reflect code complexity; when such code appears with high regularity value, it could still be easy to comprehend. They found that the use of a the regular version of a code almost always results in significantly better comprehension than other versions. Specifically, they created a controlled experiment, trying to isolate the effect of regularity. The experiment had dozens of participants. Each participant was given a booklet with code and questions and they timed each answer they had. Participants were given 3 programs (in a random variant, with high regularity or low LOC or low MCC), and were asked 3 questions about each program. The questions were open, one asking about a functionality, one about fixing a bug, and one about adding a new feature.

Like Jbara and Feitelson, we showed subjects code in different random
variants, asked 3 comprehension questions about it, and measured both time and correctness of each question. There are two main differences between their experiments and ours. The first is that we also analyzed the effect of experience on the results, while they did not. The second is that their answers were open, while ours were multiple-choice, meaning their measure of comprehension may be considered more accurate (without a guessing factor) but also subjective, based on the grader's understanding of the answer.

6.6 How to Measure Program Comprehension?

Siegmund and Schumann [24] introduce different confounding parameters on program comprehension and how to address them in different studies. They show that the confounding parameters may bias the outcome of comprehension studies. They identified parameters in the individual domain (individual background, individual knowledge, individual circumstances) and in the experimental domain (subject related, technical, context related, study-object related). They list 39 such parameters, including individual's culture, gender, knowledge, motivation, and experience and including time pressure, visual effort, instrumentation, technical problems, learning effects, language, ordering, and tasks.

Siegmund and Schumann suggest techniques to control these parameters, such as randomization (spread the parameter evenly, for large enough samples), matching (homogeneous groups according to a parameter), keeping the parameter constant, use the parameter as an independent variable (thus multiplying experimental groups and extending hypotheses to include parameter), and analyzing the influence of the parameter on the results.

One may consider their list of confounding parameters and the means to address them in our context: We kept the programming language constant; we normalized our results for each task separately; to address a learning effect, we analyzed our results firstly per class, and secondly separated the first question from the others; parameters like fatigue, gender, culture, reading time, and motivation were not taken into account in our study, and our technique to control them was mainly randomization, to spread them evenly between our questions (although we might not have had a large enough sample size); Finally, we addressed the programming experience parameter in two
ways, as an independent variable and as a parameter to analyze its influence on the results.

Di Penta et al. [18, 21] study the question of how to design experiments to measure comprehension. They write that it is challenging to measure program comprehension because of the need for human subjects in the experiments. They suggest that when researching the subject one needs to: (1) consider relationship between variables (those you test and undesirable ones), (2) choose subjects carefully and know their background, (3) design the experiments considering the different variables and factors, and make sure it is replicable, and (4) use tools to measure different aspects, like eye tracking, effort spent, and time passed.

Their suggestions are relevant to our research in several ways. We had a variety of subjects of different genders, countries, and experience. We considered what to measure (correctness and times), decided on additional interesting variables (like experience), and thought about their relationship. We used tools that we built (online website for questionnaire that measures times and color codes classes code). The site is easy to reuse for other similar experiments, thus in this sense the experiment is replicable. On the other hand, we did not use formal statistical models.

6.7 Conclusion

Many studies have attempted to measure comprehension based on correctness and time spent, and considered the relation to programming experience. In this sense, our approach to measuring comprehension is similar to these previous works. Unlike most previous studies that attempted to measure comprehension, we did not build our empirical study around a statistical model. We created an online questionnaire with difficult comprehension questions, forcing participants to spend time understanding the code and following execution paths. We mostly focused on measuring actual code comprehension. Unfortunately we did not have enough participants to make a stronger claim, with statistical significance, about the effect that method orders have on program comprehension.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this work we presented a study on the effect of method ordering on engineers’ code comprehension. We have defined four ordering strategies and examined their use in practice on a large body of open-source code. We then conducted a user study with 60 participants, which showed that while method orders have no significant effect on code comprehension correctness, some orders might reduce the time spent on comprehension, in particular for beginners, which are more sensitive to method orders than experts.

Further experiments are required in order to strengthen the validity of our results.

7.2 Future Work

We suggest the following future work directions. First, it is interesting to extend our work beyond Java and further to scripting and domain specific languages, to see whether and how the order of elements affects engineers’ comprehension.

Second, recent work has investigated eye movement while reading code, see, e.g., [8]. When approaching the source code of a class consisting of many methods, and trying to follow it in order to answer a complex comprehension question, method ordering may affect the sequence of scrolling up and down the text, as well as the required vertical eye movement. Our CLCT strategy may be able to minimize these. Thus, investigating the relationship between
method order and eye movement may be an interesting direction for future work.

7.3 Lessons Learned

We conclude with a discussion of lessons we have learned from doing this research.

One lesson we have learned relates to the design of the experiment. If we could start over, the first thing we would do is think about the statistical models fitting to our research questions, and design an experiment around them. In particular, we would try to minimize variables that could affect the experiment. For example, measuring only non-idle time when someone answers a question; try to use questions at the same difficulty level or find the optimal way to normalize the results per question or class; perhaps even asking a programming teacher to create the comprehension questions, as she would have more experience in doing so (however, exam or homework questions do not necessarily represent comprehension tasks that engineers face during their work). Also, to get more answers and hence possible more significant results we could have tried to publish the online questionnaire in additional places. Eventually we would analyze the results with the statistical models and hope to have results that are statistically significant.

If we had unlimited resources, we could have designed the experiment differently. First of all, a better way is a controlled experiment, where we can watch the subjects answer the questions and can monitor their times very accurately. We could also interview them during and after the comprehension tasks, and then see whether the order of methods affected the process of comprehension, not only its end result.

Second, we could present different questions with open answers: general questions like “what does this method do?”, “what output would we get given the specific input?”, or even “write code to add this specific feature”. That would have required a lot more time on our behalf (arranging this experiment and manually grading all the answers). It would have required a budget, as it would have taken each subject a lot more time to answer the questionnaire and so without adequate compensation it would be very difficult to find volunteers. In such a controlled environment and open comprehension questions, we could eliminate unwelcome factors and could have possibly measured comprehension in a better way.
Bibliography


על סידור מתודות

חיבור זה נועד וניתן לשתיير.People
M.Sc. באוניברסיטת תל-אביב, בית"ס למדעי המחשב
על ידי

יורי גפן

עבודת המחבר בועצה בהדרכתו של
ד"ר שחר מעוז

אפריל 2016
תכזיר

מוכנעי של kino מתודה מחלקה ג'אווה אינן השפעה על הסמנטיקה שלה. המונדה יולה לבחר איזה סדר מתודות היא מעדיפה. אילו קונים יציאת קוד בונג לסדר מתודות נפוסט בגול, אם בכלל? האם ישנו סדרי מתודות הておくים מאחרים בכר שמהו מנוקד על הבת kod? האם סדרים טובים ינימינ לחשב וליישום?

אוטומטיית?

שבועה זו אנחת מנסים לגעות שלשלות שלל. התוילה, אנו מציינים מחקר על סדרי מתודות בקורפוסי גול של פוריקיטס של קוד פיתוח, בהן אנו מдержан מנ➽יר נפוץ. הנachte, אנו מציינים ארבע אסטרטגיות של��יון מתודות, שיתוק ב WebDriver לסדר מח Klan פיתוח באימוץ תוספ לאלקילס שיאנו מסקקים כוש של 아גרון קוד מח Klan פיתוח כאטרקטיבי (refactoring). לברו, אנו מציינים את התראות של מחקר משתחמות, אשר מעריך את השפעה של אסטרטגיות הסדרי שלפ על הבת kod של המהדור.

במונבון של קומת והDoctrine שלושה במקתית.