The Order of Things: How Developers Sort Fields and Methods

Benjamin Biegel, Fabian Beck, Willi Hornig, Stephan Diehl
University of Trier
Trier, Germany
Email: {biegel,beckf,s4wihorn,diehl}@uni-trier.de

***Abstract*—In source code files, fields and methods are arranged in linear order. Modern programming languages such as Java do not constrain this order—developers are free to choose any sequence. In this paper we examine the largely unexplored strategies developers apply for ordering fields and methods: First, we use visualization to explore different ordering criteria within two open source Java projects. Second, we verify our observations in a metric-based analysis on an extended set of 16 projects. Third, we present the results of a survey that reflects the opinion and applied ordering strategies of 52 developers. 87% of the participants agreed that ordering of fields and methods is meaningful or important. Our results suggest that there exists a set of criteria repeatedly used for ordering. Among these, the categories defined in the official Java Code Conventions appear to be the primary ordering criterion. However, in the individual strategies of the participants of the survey, we identified 15 ordering criteria additional to the five criteria we considered in the empirical analysis.**

***Keywords*-code conventions; code navigation; linear arrangement;**

I. Introduction

"Ordering Java methods is probably the least important code style issue ever ... "[1] But to our great surprise we had to learn when discussing this issue among the authors that we apply fundamentally different ordering strategies. This raised our curiosity and motivated us to investigate whether ordering fields and methods is really that irrelevant. The first question we asked ourselves was: *In which scenarios could the ordering of fields and methods be important?* We identified the following use cases:

- Searching for a particular field or method in a class
- Searching for a certain functionality vaguely known
- Understanding a certain functionality
- Adding new fields or methods
- Following a program execution (e.g., while debugging)
- Understanding the high-level purpose of a class

The ordering of fields and methods is not vital for the above use cases, but has the potential to slow down or accelerate the work. Browsing through a source code file is a form of linear search and could be time-consuming when the fields and methods are not arranged properly. Many IDEs, however, provide help for finding particular code fragments: developers may use a search prompt or look at the code outline view (a list of fields and methods that can be sorted automatically). But still, some of those tools require a linearized representation of code entities. In the end, the order of fields and methods probably has an effect on code navigation and code comprehension.

In this paper we want to find out how developer sort fields and methods: *Are there any conventions followed by a majority of developers? What are the criteria that influence the ordering? How do developers perceive the issue of ordering?* We focus on Java projects to study these research questions. Our approach follows a *sequential exploratory strategy* [1] where first hypotheses are derived from an explorative analysis, and second those hypotheses are evaluated in a quantitative experiment. In particular, the study consists of a visual analysis on two open source systems (Section III), an empirical analysis on 16 open source systems (Section IV), and a survey with 52 participating developers (Section V). Our data from the empirical analysis and the survey is available online for replication: http://www.st.uni-trier.de/icsm12/.

II. Ordering Approaches

The ordering of code entities in source code documents has not yet been empirically explored. There exist, however, different paradigms to deal with the issue of ordering. While the ordering of fields and methods is not constrained in many programming languages, some require that a method must be declared before it is called (e.g., Pascal, C, and C++). Therefore, header files are used to aggregate the declarations and method definitions can be again ordered as desired.

Code entities might get grouped according to high-level categories—the official *Java Code Conventions (JCC)* [2] discern between four categories of attributes and method as depicted in Figure 1: **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**. The categories should appear in the order as listed. They form a *primary ordering criterion*.

As the *secondary ordering criterion*, according to the JCC, class and instance variables should then be sorted by

[1]This is the first sentence of a question titled "What is the most common way of ordering Java methods?" by *JRoberts* posted November 12, 2010 at *stackoverflow.com*, which received five non-agreeing answers – (last retrieved: 2012-01-27) http://stackoverflow.com/questions/4166251/what-is-the-most-common-way-of-ordering-java-methods

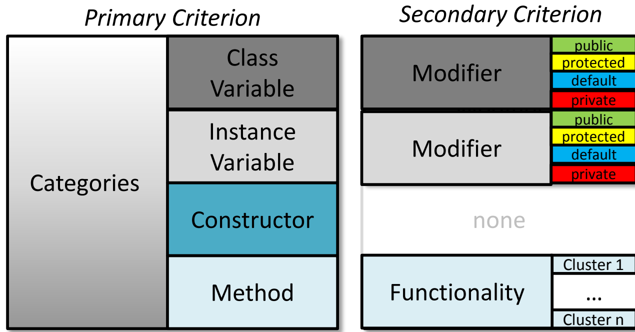

Figure 1. Ordering criteria according to the Java Code Conventions (JCC).

visibility in the order *public*, *protected*, *default*, and *private*; recommendations for the ordering of variables below this level are not provided. In contrast to variables, the coding conventions postulate for methods—somewhat vague—that they *"should be grouped by functionality rather than by scope or accessibility."*

As demonstrated by Class Blueprints [3], different ordering criteria can be combined to more elaborate, generic categories. The approach discerns five layers: an initialization layer containing constructors and init-methods, an interface layer containing methods of the public interface of the class, an implementation layer containing the private methods, an accessor layer containing getters and setters, and an attribute layer containing the fields. These layers are used to visualize and understand the characteristic features of classes.

Also call dependencies can be considered for deriving an order: Martin [4] advocates the idea of placing the definition of a method immediately after it is called for the first time (*stepdown rule*). This approach leads to a top-down ordering of methods from a high to a low level of abstraction, and so should *"[. . .] help[s] the program read like a newspaper article"*.

There exists tool support for ordering fields and methods. For instance, the Eclipse IDE provides functionality to automatically sort fields and methods in source code documents: Here, static and non-static entities are discerned, as well as fields, initializers, methods, and constructors. These definitions are similar to the JCC, though vary slightly. As an optional secondary criterion, entities in the same category can be ordered by visibility. Below category or visibility level, entities are automatically sorted lexicographically. Similar functionality is available as plug-ins in other IDEs such as Visual Studio or IntelliJ, and in standalone tools like JCSC.

Additional to improving the order, code navigation can be eased by other means. DeLine et al. [5] use thumbnail views of the code layout as extended scrollbars to navigate through the code. Mylyn [6] computes a degree of interest metric and can hide those fields and methods the developer probably does not need for the task at hand. Desmond and Exton [7] present a code exploration technique that embeds related code from a different file into the current editor on demand.

Source code, however, does not need to be organized in documents containing long lists of methods—an alternative are package and class browsers in Smalltalk IDEs such as Squeak. Packages, classes, and methods are organized in side-by-side panes; the editor pane does not show whole source code documents, but just the code of the currently selected method. Nevertheless, ordering is also relevant for such kinds of tools because fields and methods have to be listed linearly somewhere for selection.

A visual ordering approach is to map code entities onto a two-dimensional map. Such a map can be integrated into an IDE and used for code comprehension and navigation as Kuhn et al. propose [8]; they arrange code entities according to their semantic similarity, which is derived from the vocabulary used in the code. Code Canvas [9] demonstrates how a map metaphor can be employed as the basic outline of an IDE. Code Bubbles [10] is an IDE prototype that uses a desktop metaphor and organizes code entities such as methods in bubbles, which can be grouped freely on the desktop.

III. Explorative Analysis

As the first step of the *sequential exploratory strategy* , an explorative analysis targets at deriving hypotheses on how attributes and methods are ordered in software projects. To this end, we analyze two open source Java systems—namely, *JUnit* (a unit testing framework) and *Stripes* (a web framework). In the focus are original ordering criteria that are either recommended by coding standards or are enforced by software development tools. Simple visualizations are used to generate and illustrate hypotheses (although we tried to optimize the color scales with respect to black-and-white prints, we still recommend reading a colored version of the paper to see full details). Along the way of visual exploration, we introduce the data acquisition procedures that are necessary for studying the different ordering criteria.

A. JCC Categories

First, we investigate whether developers follow the official Java Code Conventions (JCC) as a primary criterion. As described in Section II, the JCC split code entities into four categories; we assign a color to each category for visualization purposes: class variables (black), instance variables (dark cyan), constructors (orange), and methods (light orange). For inspecting the compliance of a large set of classes with this standard, we propose a simple, space-efficient visualization: Each class is represented by a vertical bar that consists of stacked uniformly sized boxes; each box represents one of the entities and is colored according to the type of the entity.

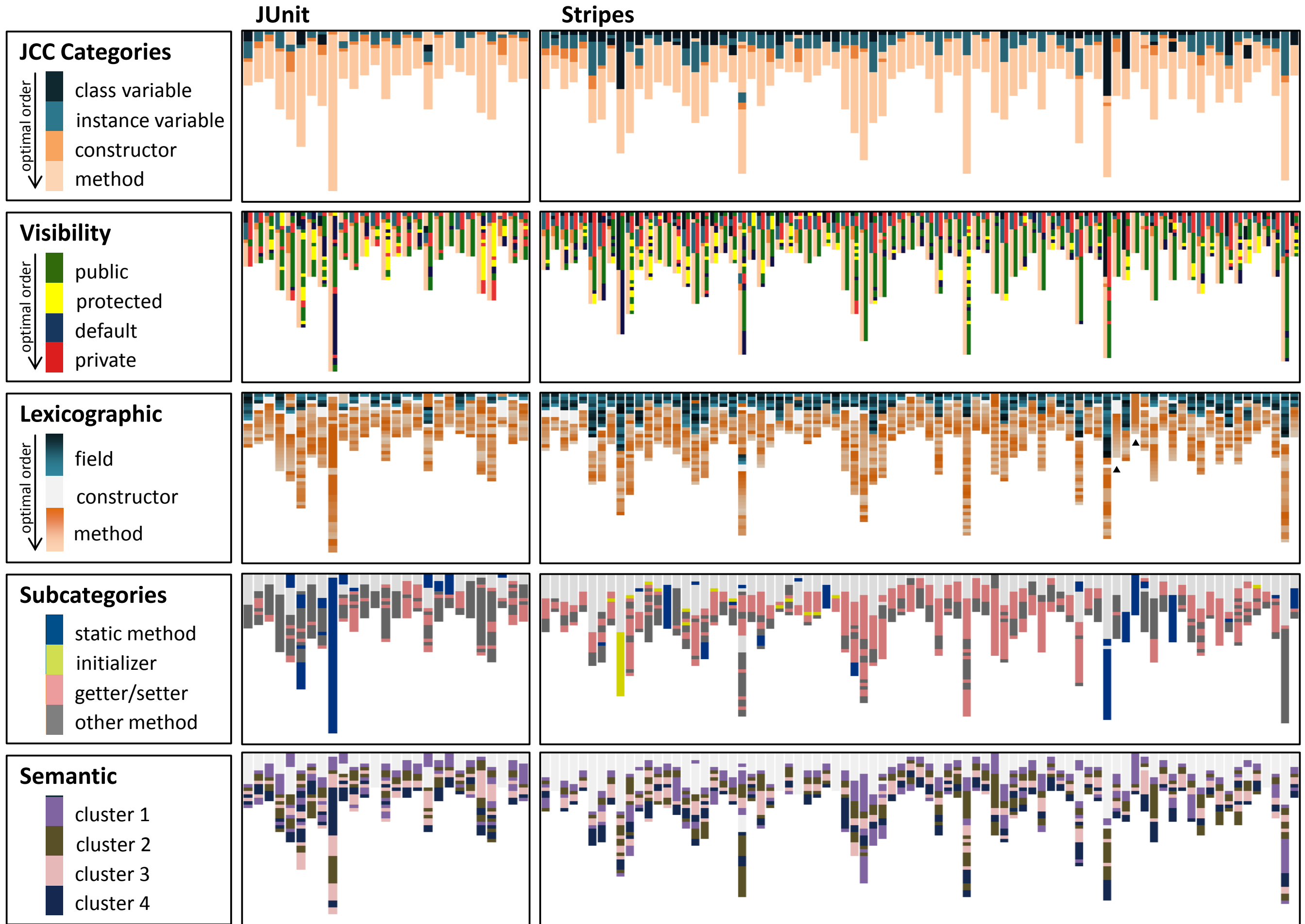

Figure 2. Visualization of the applied ordering criteria in *JUnit* and *Stripes*: JCC categories, visibility, lexicographic, subcategories, and semantic.

Figure 2 shows such a bar chart visualization for the two sample systems, the first row depicting the JCC Categories. We excluded small classes with less than ten entities because the order of the entities does not become apparent if there are only few entities per category if any. Due to space limitations for the visualizations, we also skip classes with more than 50 entities.

What we observe is that many classes across the two projects show a bar in perfect order—they match the JCC categories including the proposed ordering of the categories. Only few classes show a slightly varying pattern, but still the entities seem to at least roughly follow the specified grouping. From these observations we derive the following hypothesis, which we want to test in the subsequent empirical analysis.

Hypothesis 1 (JCC Categories)**.** *(a) The categories defined in the JCC are used as a primary sorting criterion in the majority of classes. (b) They are used at least as a partial grouping criterion in nearly all classes.*

This assumption covers only the primary ordering strategy as defined by the conventions; secondary strategies are discussed in the following. Please note that we distinguish between *sorting criteria* (entities must have the correct order) and *grouping criteria* (entities need to be grouped correctly, but the order of the groups is not considered). If we refer to both, we use the term *ordering criteria*.

B. Visibility

Ordering entities by visibility is defined as the secondary ordering strategy for attributes in the JCC in the following sequence (we again assign colors): public (green), protected (yellow), default (dark blue), and private (red). This criterion is considered as a primary or secondary sorting strategy in Eclipse and JCSC.

Visualizing visibility modifiers of attributes and methods similar to the categories could help to detect whether those are used as a primary ordering criterion. To also detect applied secondary ordering criteria, we represent each class as a split bar in Figure 2 (second row), the left part showing

the JCC categories, the right part reflecting the visibility modifiers.

The code entities seem to be ordered less according to visibility modifiers than to the JCC categories. Searching for strictly grouped classes at the level of the primary ordering criterion (without considering the proposed order of visibility modifier), we only find few examples for JUnit, and some more for Stripes. Judging whether visibility is used as a secondary ordering criterion for variables is more difficult because many classes only include few variables, often also having the same visibility. In contrast, the classes tend to have more methods than fields and those also vary in visibility; some strictly grouped method blocks can be observed. But, although visibility does not have high rates of strictly grouped classes, the majority of classes seem to be grouped by this criterion to some extent—visibility modifiers mostly appear in larger blocks.

Hypothesis 2 (Visibility)**.** *(a) In some cases the visibility is used as a strict grouping criterion. (b) But entities are grouped partially with respect to visibility in the majority of classes.*

C. Lexicographic

For lexicographic ordering entities are indexed according to their position in a lexicographically sorted list. This ordering is unambiguous and hence can easily be automated. Although this approach is not proposed by the JCC, Eclipse and other development tools implement it. For lexicographically ordering fields and methods, we applied the `String.compareToIgnoreCase(String)` method of the standard Java API; method parameters are ignored and hence polymorphic methods are assigned the same index.

As already observed, the JCC categories seem to be applied as the primary ordering criterion. Hence, we mainly check whether lexicographic sorting is used as a secondary criterion and investigate the ordering of fields in separation from the ordering of methods. We visualize both in the same diagram (Figure 2, third row). The color mapping assigns a color ranging from black to cyan to each field, and a color ranging from orange to light orange to each method; darker colors indicate entities with a smaller lexicographic index. Constructors are colored in light gray.

What we observe for fields as well as for methods is that only very few classes are ordered perfectly according to this criterion—perfect lexicographic orderings create pure dark-to-light color scales. Such perfect scales can only be found for some entities in the Stripes project: We marked two classes with a perfectly sorted method block with small black triangles in Figure 2. In some more, but still rare cases, parts of the blocks of fields or methods are sorted—explanations could be an inherited list of methods automatically added in sorted order by the IDE or other sequences of generated code.

Hypothesis 3 (Lexicographic)**.** *(a) Lexicographic ordering is rarely used as a strict sorting criterion for fields and for methods, neither as a primary criterion nor as a secondary one. (b) In some classes, however, it is considered partially.*

D. Method Subcategories

Methods can be classified into a set of generic subcategories. For instance, getters and setters are very special types of methods, which frequently occur in many classes. Inspired by the subcategories used in Class Blueprints [3] and proposed by the automated ordering algorithm in *Eclipse*, we distinguish between the following set of method subcategories (no particular order): static methods of every kind (blue), non-static initializers (yellow), non-static getter and setter (pale red), and other non-static methods (gray). Initializers are non-static initialization blocks or non-static methods that start with the string "init". Getters are identified as methods having a non-void return type; their name starts with "get" or "is", followed by a capital letter. In contrast, the name of setters starts with the string "set" followed by a capital letter; setters have at least one parameter.

As Figure 2 (fourth row) indicates, the distribution of the above subcategories across classes seems to be less uniform than the groups in other criteria. While most of the classes have getter/setter methods and other methods, only few contain static methods and initializers. In both systems, a considerable number of method blocks are strictly grouped by the method subcategories. Beyond that, we also observe larger blocks of methods belonging to the same subcategory in the other classes.

Hypothesis 4 (Method Subcategories)**.** *In a considerable amount of classes, the defined method subcategories are applied as (a) a strict criterion to group methods as well as (b) a partial criterion.*

E. Semantic Similarity

Finally, we look at the grouping by functionality, which is propagated by the JCC for sorting methods. But it is hard to tell what the term *functionality* exactly refers to; also the conventions do not provide further information. Inspired by other approaches that group code entities by their similarity in purpose, we propose an approach based on the vocabulary used in the identifiers and comments of methods. Methods of similar vocabulary are rated as semantically similar. The similarity can be used to cluster the methods into different groups, which allows studying those clusters using our visualization technique.

Our approach is similar to the approach of Kuhn et al. [11]: From the source code of each method including possible preceding comments, we extracted all words, also splitting camel-case identifiers. From this list of words, Java keywords and English stopwords are removed and stemming is applied (*Porter algorithm*). Then, a method is described as a vector, each dimension representing the occurrence of a

particular word. Instead of the word frequency, we consider the *term frequency–inverse document frequency*, which is a widely applied information retrieval technique; it balances the importance of each word so that words occurring in many methods become less relevant.

A *hierarchical agglomerative clustering* algorithm based on the cosine similarity of the vectors is used to create semantic clusters of methods. To be consistent with the number of groups in the other ordering criteria, we cut the hierarchical clustering into four clusters for each class. We consider only those classes for clustering that have a single cohesive method block consisting of at least five entities.

This approach can only be considered as a heuristic to analyze the grouping by functionality, or *semantic grouping* as we call it in the following. But in general, the definition of this criterion is too vague to be measured exactly. We checked a set of clusterings manually and found that the results generally matched our intuition for semantically grouping.

Figure 2 (last row) shows the outcome for the two sample systems, each cluster represented in a different color. Please note that the numbering of clusters is arbitrary; equal cluster indices across different classes do not have any meaning. A first observation is, that the cluster sizes seem to be evenly distributed; no giant clusters affect the results. Due to the heuristic nature of the clustering approach and in contrast to the previous criteria, strictly grouped classes do not have a particular meaning beyond being well-ordered. Nevertheless, it can be analyzed how many classes are partially ordered: We observe a reasonable number of such classes mainly in Stripes; but also in JUnit we find some classes having cohesive blocks of methods belonging to the same cluster.

Hypothesis 5 (Semantic)**.** *In many classes with a single cohesive method block, methods are partially grouped with respect to their semantics.*

IV. Empirical Analysis

As the second step of the *sequential exploratory strategy*, we provide quantitative evidence in this section to support our hypotheses. For this purpose, we introduce two metrics to measure the impact of sorting and grouping.

We analyze 16 open source Java projects of varying sizes and from different areas of application. These projects are listed in Table I together with the number of classes considered for the analysis. Since enumeration types and interfaces are not suitable for our applied semantic similarity measures, we ignore these structures. All classes with fewer than 10 entities are also skipped, first because the order in small classes is less important, and second to minimize the risk of getting perfectly ordered classes by accident. Despite these limitations, we inspect 50% of all classes, which altogether contain about 85% of all fields and methods.

Table I
Properties of the analyzed software projects.

Project	Description	Version	#Classes
Checkstyle	coding conventions	5.1	88
Cobertura	test coverage	1.9.4.1	25
CruiseControl	continuous integration	2.8.4	150
iText	*PDF* library	5.0.5	219
JabRef	*BibTeX* management	2.6	201
JEdit	text editor	4.3.2	240
JFreeChart	chart library	1.0.13	339
JFtp	*FTP* client	1.0	29
JHotDraw	GUI framework	7.6	296
JUnit	regression testing	4.5	28
LWJGL	gaming library	2.7.1	150
PMD	code problems	4.2	110
Stripes	web framework	1.5.5	88
SweetHome3D	interior design	3.1	117
TV-Browser	program guide	2.7.6	438
Wicket	web framework	1.2.2	202

A. Sort and Group Metric

In this work we consider order criteria, which are based on sequences as well as groups. Therefore, we define two order metrics to measure the degree of ordering for a set of entities. Before testing whether a sorting criteria was considered in a sequence of entities, we index the entities according to their position in a sorted list. The same is true for testing whether a grouping criteria was applied to a set of entities. In this case we assign the entities to their belonging groups. For these purposes, we define the **reference index**, which returns either the corresponding sorting position or group number for an entity.

Definition 1 (Reference Index)**.** *Let E be the set of all entities and $\varphi : E \to \mathbb{N}$ a function that maps entities to natural numbers. We call $\varphi(x)$ the* ***reference index*** *of x.*

For measuring order, we look at sequences of correctly ordered entities and count pairs of entities that are in the right order according to the reference index. The **sort metric** checks for an entity whether the reference index of the consecutive entity is higher (or at least equal). A value of 1.0 corresponds to a perfectly sorted entity set.

Definition 2 (Sort Metric)**.** *Let $e = (e_1, \ldots, e_n)$ be a sequence of at least two entities, then the* ***sort metric*** s *is defined as*

$$\mathrm{s}(e) := \frac{|\{i : \varphi(e_i) \leq \varphi(e_{i+1})\}|}{|e| - 1}$$

The **group metric** is similar to the sort metric: It counts all pairs of entities with the same reference index. A value of 1.0 corresponds to a perfectly grouped entity set.

Definition 3 (Group Metric)**.** *Let $m_e := |\{k : \varphi(e_i) = k\}|$ be the number of different indices covered by a sequence of*

at least two entities $e = (e_1, \ldots, e_n)$, then the ***group metric*** g *is defined as*

$$\mathrm{g}(e) := \frac{|\{i : \varphi(e_i) = \varphi(e_{i+1})\}|}{|e| - m_e}$$

when $2 \leq m_e < |e|$.

B. Partial Sorting and Grouping

The hypotheses raised in Section III make assumptions on the frequency of partially sorted or grouped classes. While it is simple to find strictly ordered classes, it is much more difficult to judge when a class is partially sorted or grouped with respect to a particular criterion. Just defining a certain threshold for the respective metric and counting those classes above this threshold would be misleading because the metric value does not only depend on the ordering but also on the number of clusters m_e and their size distribution. For instance, when organized randomly, a class consisting of only private code entities plus an additional public entity usually reaches higher cluster metric values than a class with more evenly distributed visibility modifiers.

A better way of identifying a partially sorted or grouped class is to contrast its metric value to an equivalent class consisting of the same code entities in a random order. If the metric value of the original class is higher, this is an indicator that the particular criterion has played a role. But the α-error, the probability to judge a randomly ordered class as ordered by mistake, is 50%. If a repetition of the procedure yields further random metric values smaller than the original metric value, our confidence increases that the original class is partially sorted or grouped. Targeting an α-error of 5%, the original order must be best among 20 orders, 19 created randomly. Hence, we perform this random experiment for every class: we compute its metric value; shuffle the code entities 19 times; and if the original value is the highest, we reject with sufficient confidence that the original ordering was created randomly—it is rated as *partially ordered*.

This procedure is independent of the metric and hence works for the sort metric as well as the group metric. Though statistically founded, it is only a heuristic to count the partially ordered classes: On the one hand, we still have an α-error of 5% to rate purely randomly ordered classes as partially sorted or grouped. On the other hand, partially ordered classes may not be assessed as such because their metric values are too close to the random ones (β-error). In the end, this approach provides a form of estimated lower bound of the number of partially sorted or grouped classes.

C. JCC Categories (Hypothesis 1)

The results of the empirical analysis are summarized in Figure 3. Every boxplot of this figure shows the distribution of percentage values per project, each reflecting the ratio of sorted or grouped classes according to a particular ordering criterion. To better understand the results of the empirical analysis, we recommend to use Figure 3 continuously in the following subsections.

Primary Criterion—In the explorative analysis, the JCC categories are identified as the most dominant ordering criterion. This issue is also represented in the high metric values. As we can see in the *strictly part* of Figure 3 (right diagram), the upper two boxplots are much higher in the scale than the others. This means that most projects have a high ratio of strictly ordered classes according to the JCC categories. In detail, more than 78% of all classes are strictly sorted and 81% grouped in half of the projects. The small difference of the medians suggests that, whenever a class is grouped, it is also sorted with respect to the JCC categories. With values below 60%, only *JEdit* and *JFtp* are outliers and do not strictly apply this criterion as much (in the boxplots marked by circles). However, almost all analyzed projects meet Hypothesis 1(a): *The categories defined in the JCC are used as a primary sorting criterion in the majority of classes.*

In further analysis we find that over 80% of *all* classes are ordered partially by the JCC categories. Half of the projects consist of more than 90% partially grouped classes. This fact supports Hypothesis 1(b): *The JCC categories are used at least as a partial grouping criterion in nearly all classes.* In addition, more than 95% of all classes in half of the projects are partially sorted. Thus, we can further specify this result: The JCC categories are used at least as a partial *sorting* criterion in nearly all classes.

D. Visibility (Hypothesis 2)

Primary Criterion—Our results report a median of 0% for visibility as a strict primary sorting criterion. *JEdit*, however, stands out with a ratio of 27%. In much more cases the visibility is applied strictly as a grouping criterion. A median of 22% supports Hypothesis 2(a): *In some cases the visibility is used as a strict grouping criterion.*

In contrast, with a median of 68%, a considerably larger proportion of classes is *partially* grouped by visibility. Thus, we agree with Hypothesis 2(b): *Entities are grouped partially with respect to visibility in the majority of classes.* It is also worth mentioning that the median of partially *sorted* classes is still 48%.

Secondary Criterion—For the analysis of visibility as a secondary ordering criterion, we subdivide the classes with respect to the JCC categories into three parts—class variables, instance variables, and methods. In each case we only consider those classes that have at least five entities belonging to the particular category. In addition, the entities have to form a single cohesive block. These restrictions lead to a decreased number of classes. To still get meaningful results, we only include those projects in the analysis which have at least 20 classes fulfilling the requirements.

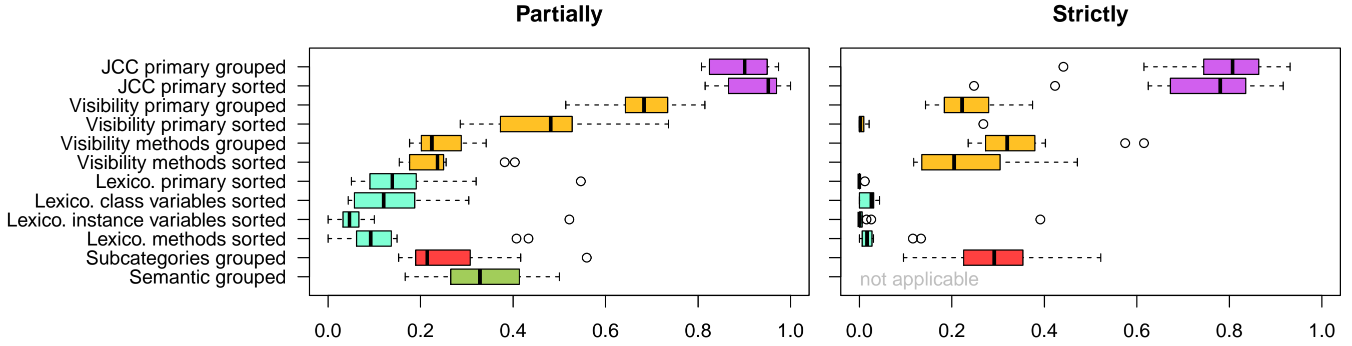

Figure 3. Percentage of classes per project which are sorted or grouped according to the given ordering criterion (whiskers with maximum 1.5 IQR).

As observed in the explorative analysis, many classes have only a small set of fields, which mostly have the same visibility (e.g., on average 83% of all instance variables are private). After filtering the projects, only three projects contain enough classes to be selected for our analysis. Thus, for the visibility, a separate analysis of cohesive class or instance variable blocks is not reasonable. Nevertheless, this insight is interesting in combination with the metric values of the visibility in method blocks: More than 24% of cohesive method blocks are partially sorted and 22% are partially grouped in half of the projects. Surprisingly, a much larger proportion of whole classes is partial grouped than method blocks. It seems that the order is influenced more by the JCC categories than by the visibility. In particular, the high amount of partially ordered classes could be based upon several sequences of fields with the same visibility. We assume that the developers were not aware of following this ordering criterion as much. This issue can also be supported by a higher median of 32% for *strictly* grouped classes—*Checkstyle* with even 62% and *PMD* with even 57%. Nevertheless, our measure results fully agree with Hypothesis 2, even though the order seems to be influenced by the JCC categories.

E. Lexicographic (Hypothesis 3)

Primary Criterion—The test on lexicographic order as a strict primary criterion provides clear results: In total only one single class (in *Stripes*) is sorted strictly.

Secondary Criterion—We use the same categories and the same filtering approach as described in the previous section. Since every field has an unique sorting index, the filtering only depends on the number of fields in a class. We consider 6 projects for inspecting cohesive class variable blocks and 13 projects for cohesive instance variable blocks. Whereas altogether the fields rarely follow the lexicographic order strictly, a large proportion of 39% of strictly sorted instance variables in *Wicket* is remarkable. Furthermore, this project and *Cobertura* both use the sorting criterion strictly on 12% and 13% respectively of their cohesive method blocks. All other projects have values between 0% and 3%. In summary, all these findings support Hypothesis 3(a): *Lexicographic ordering is rarely used as a strict sorting criterion for fields and for methods, neither as a primary criterion nor as a secondary one*.

Analyzing partial sorted classes, the medians increases to 14% for whole classes, 12% for class variables, 5% for instance variables, and 9% for methods). Again *Cobertura* (32%) and *Wicket* (55%) stand out with many partial sorted classes. Thus, our measures also support Hypothesis 3(b): *In some classes, however, lexicographic ordering is considered partially*.

F. Method Subcategories (Hypothesis 4)

Secondary Criterion—We measure in half of the projects at least 29% of classes, which are strictly grouped by the defined subcategories. We find that the subcategories are used differently in the analyzed projects. For example, in *JFtp* only 10% of the classes are strictly grouped, and hence, the subcategories are less considered. In contrast, *JFreeChart* has a ratio of 52% strictly grouped classes and *Stripes* a ratio of 50%. In summary, these results agree with Hypothesis 4(a): *In a considerable amount of classes, the defined method subcategories are applied as a strict criterion to group methods*.

By taking all partial grouped classes into account, the median decreases to 21%. This fact could give a hint that some classes are grouped accidentally, and thus, the developers possibly were not aware of using this grouping criterion in some classes. However the metric values still support Hypothesis 4(b): *In a considerable amount of classes, the defined method subcategories are applied as a partial criterion*.

G. Semantic (Hypothesis 5)

Secondary Criterion—As mentioned in Section III-E, the semantic criterion can only be measured as a heuristic—strictly grouped classes do not have a distinguished meaning. Nevertheless, partial grouped classes provide evidence that

	Grouping criteria	1	2	3	4	5
1	JCC primary	1.00	0.35	0.09	0.10	0.00
2	Visibilty primary	0.35	1.00	0.61	0.21	0.14
3	Visibilty methods	0.09	0.61	1.00	0.24	0.23
4	Subcategories methods	0.10	0.21	0.24	1.00	0.26
5	Semantic methods	0.00	0.14	0.23	0.26	1.00

Figure 4. Mean correlations between grouping criteria.

semantics are used as a grouping criterion. In half of the projects we find over 33% of partially grouped classes by semantics. Larger proportions can be found in *JFreeChart* (50%), *SweetHome3D* (49%), or *iText* (44%). Even if we consider the projects with the smallest proportions, e.g. *Cobertura* with 17%, we can still establish a rather large amount of impacted classes by semantics. Therefore we agree with Hypothesis 5: *In many classes with a single cohesive method block, methods are partially grouped with respect to their semantics*.

H. Discussion

In summary we observe a majority of classes being arranged according to the categories defined in the JCC as a primary criterion. The four other tested criteria were not applied as strict as those categories, neither as a primary nor as a secondary criterion. Nevertheless, our results suggest that the majority of classes is at least partially grouped by visibility modifiers. In cohesive method blocks, grouping by method subcategories and by semantic similarity seems to play a major role.

Observing a criterion in the code does not necessarily mean that it was directly applied by the developers, but could as well be a side-effect of applying another. Therefore, it is important to discuss correlations between the criteria. We look at the grouping criteria and compute for each project and criterion a vector that reflects each class in a normalized metric value (the difference between the order metric and the best random ordering metric value). These vectors are compared using the Pearson correlation for each project. Figure 4 averages the correlations over all projects in a correlation matrix.

The only strong correlation (0.61) can be found between visibility as a primary and a secondary criterion, which is not surprising because both are inherently related. The other correlations seem to be much weaker and hence do not significantly affect the results of this study: The moderate correlation of 0.35 between JCC categories and visibility (both as primary criterion) might be a consequence of the fact that entities of the same category often seem to have the same visibility (also compare to Figure 4, second row). Hence, the quite high percentage of partially grouped could be partly explained by this correlation. For the metrics on method block level, we observe a weak relationship between visibility, method subcategories, and semantic similarity (values between 0.23 and 0.26), which indicates that these criteria are not totally orthogonal. An analogous analysis was performed on the sorting criteria but did not show any further considerable correlations.

V. Survey

To complete our analysis of ordering strategies, we conducted a small survey among developers: On the one hand, we were interested in whether developers are aware of ordering fields and methods and how they rate its importance. On the other hand, we wanted to find out concrete sorting strategies applied.

We sent the survey consisting of seven questions by e-mail to colleagues in research and industry as well to students and asked the recipients for further distributing it among their colleagues. We also posted it in social networks. The survey could be filled in by simply answering the e-mail or by using an alternative (but equivalent) web-form.

We received 55 answers and considered 52 of these for the analysis (two accidental double-submissions as well as a participant declaring he/she did not understand a question were excluded). Figure 5 presents the results of the six non-free-text answers. The diagrams in the top row describe meta-information about the participants: Among the 52 participants, 17 were from industry and 23 from research. 63% name Java as one of their primary programming languages; 88% regard themselves as advanced or expert Java programmers.

Nearly all of the participating developers think about ordering fields and methods when programming: 92% consider this issue at least rarely, and still 55% frequently or always. It is interesting to note that the percentage of developers who always think about ordering is only 4% among the non-expert Java developers, but 46% among the developers who rate themselves as Java experts. But although people seem to worry about the problem of ordering, automatic ordering is not often applied—88% rarely or never use such tool support. Considering the developers' subjective opinion, the ordering of fields and methods is rated as meaningful or important by 87% of the participants. Nobody, however, goes as far as naming the issue of ordering "crucial".

We also asked the participants to describe their ordering strategy and provided a free text form. Since we did not want to bias the developers, we did not give any hints at ordering strategies that could be used. We tagged the answers we received using the five criteria that we already covered in the empirical analysis. A described strategy is assigned to a criterion if both are similar, not necessarily equal; in many cases a strategy needed to be tagged with multiple criteria.

Figure 6 summarizes the tagged strategies. First of all, we observe that each criterion was at least named by 6 participants—developers seem to be aware of the ordering criteria we investigated. The most frequently addressed issue (38% of participants) were categories similar to those in the

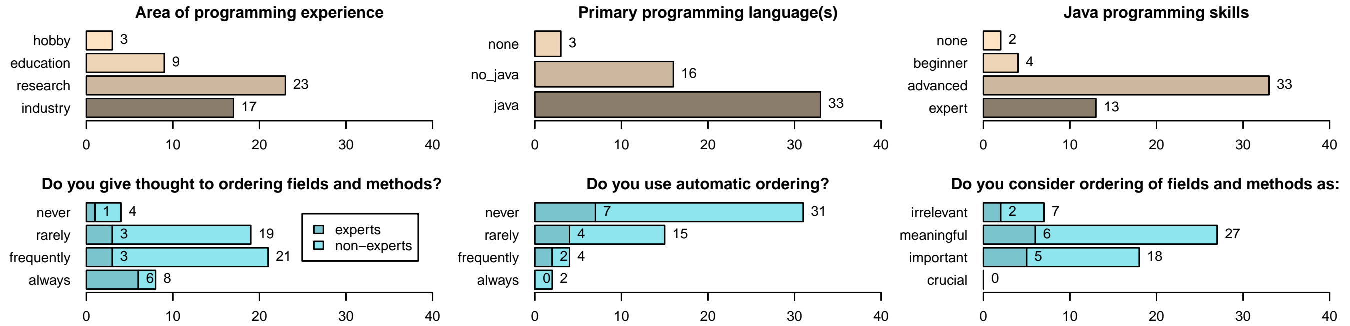

Figure 5. Summary of the survey results; top row: questions providing meta-information on the participants; bottom row: questions on ordering methods and fields (shaded bar: opinion of the participants rating themselves as Java experts).

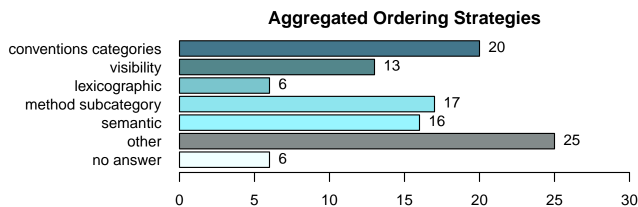

Figure 6. An aggregated list of identified ordering strategies described by the participants (multiple tags per participants possible).

Table II
ALTERNATIVE ORDERING STRATEGIES APPLIED.

Strategy	Description
field data type	group fields of the same type together
size	prioritize methods by their size
override	group overridden methods together
specialization	group methods according to their degree of specialization
importance	prioritize fields or methods by their importance
fields to accessors	group fields and their respective accessors (getter/setter)
fields to methods	group fields and methods that use the fields
call locality	group methods that call each other together
internal–external	group internally and externally used methods together
frequency of usage	prioritize methods by the frequency of usage
declare before use	sort methods so that the callee is declared before the caller
execution sequence	sort methods in the order of execution
evolutionary	group those entities together that are inserted together
authors	group methods according to their authors
ignore	do not order fields and methods

JCC, which matches our results from the empirical analysis (Hypothesis 1). But also method subcategories (33%), grouping by semantic (31%), and grouping by visibility (25%) played a major role.

Beside the criteria we studied in detail, the participants used a wealth of other criteria. Among the alternative strategies, which are named by 25 participants, we identified 15 further unique criteria listed in Table II. Summarizing these strategies, some people group fields and methods by specific characteristics of the respective entity: the data type, the size, the degree of specialization, the importance, or whether a method overrides another. In most of the alternative strategies, however, usage and call relationships are important: fields and methods are grouped together that use each other, external calls and their frequencies are considered, or the execution sequence determines the ordering. Moreover, criteria related to the evolution of the system are named by some participants. Two developers, however, explained that they do not apply a particular order, but always use tools to find methods.

As a result of the survey, the applied strategies are far from being consistent. In contrast, the empirical analysis suggests that the JCC categories seem to be widely applied in practice. A simple explanation for these seemingly contradicting results, however, could be that the conventions categories form a sort of consensus among the developers, but the individual ordering criteria are much more complex than just a weak grouping by category.

VI. Threats to Validity

To discuss threats to validity, we distinguish *conclusion validity*, *internal validity*, *construct validity*, and *external validity* [12] .

Conclusion Validity—*Is there a relationship between the actual order of entities and the ordering criteria?* We measure the percentages of classes that are strictly or partially sorted or grouped according to the ordering criteria. But there exists only a relationship if it can be rejected with sufficient confidence that these numbers were a product of chance. By taking only classes into account that consist of at least ten relevant entities, an accidentally created strict order is sufficiently unlikely. For estimating the percentage of partially sorted or grouped classes, we took a conservative approach and only count those confirmed as ordered in a random experiment with an α-error of 5%.

Internal Validity—*Is the observed relationship causal?* A high percentage of ordered classes does not proof that the particular criterion was really applied—a different crite-

rion having similar effects could be responsible as well. Analyzing the correlations, however, did not reveal any strong dependencies between different groups of criteria. Moreover, the criteria we studied in detail are mentioned by a reasonable number of participants in the survey.

Construct Validity—*Do we measure the constructs as intended?* We reflect the ordering criteria in the different reference indices that we assign to the entities. For some criteria like the JCC categories or the visibility, the assignment is straightforward. But already for the lexicographic ordering and the method subcategories, there could be different definitions—we tried to select common definitions. Even more problematic is the semantic clustering, where the assignment of semantic clusters can only be considered a heuristic. Moreover, the sort and group metric may not appropriately measure the degree of order.

External Validity—*Are the results generalizable?* The studied systems are partly representative as they span a wide set of different project types, but are still all open source systems written in Java. Since ordering fields and methods also depends on language constructs and limitations as well as on code conventions, our results of the empirical study cannot be directly transfered to other programming languages. In contrast, although we specifically asked for the degree of experience in Java, the survey was not limited to Java developers and provides a broader picture. The group of participants, however, is selective and cannot be considered representative for general software developers.

VII. Conclusion

The presented work is the first study on ordering fields and methods. The results show that the main categories defined in the Java Code Conventions (JCC) are widely implemented in the studied software projects as a primary ordering criterion. At the level of secondary ordering, however, the applied criteria seem to be more diverse. Our results also revealed a certain gap between the importance of the topic confirmed by the survey and the diverse individual strategies applied by developers. A first step towards closing this gap could be to make developers more aware of this issue—development environments may identify and highlight existing orders. The next step would be to better support developers in applying a certain ordering strategy, for instance, by providing semi-automatic ordering tools.

> *Thus, in every culture, between the use of what one might call the ordering codes and reflections upon order itself, there is the pure experience of order and of its modes of being.*
>
> —Michel Foucault: The Order of Things [13]

References

[1] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research guide to advanced empirical software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer London, 2008, ch. 11, pp. 285–311.

[2] Sun Microsystems. (1999, April) Code conventions for the Java™ programming language. [Online]. Available: http://www.oracle.com/technetwork/java/codeconventions-141855.html#1852

[3] S. Ducasse and M. Lanza, "The class blueprint: visually supporting the understanding of glasses," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 75–90, 2005.

[4] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[5] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson, "Code thumbnails: Using spatial memory to navigate source code," in *Proc. of the Symposium on Visual Languages and Human-Centric Computing VL/HCC*. IEEE, 2006, pp. 11–18.

[6] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proc. of the International Conference on Aspect-Oriented Software Development AOSD'05*. ACM, 2005, pp. 159–168.

[7] M. Desmond and C. Exton, "An evaluation of the inline source code exploration technique," in *Proc. of the Annual Workshop of the Psychology of Programming Interest Group PPIG'09*, 2009.

[8] A. Kuhn, D. Erni, and O. Nierstrasz, "Embedding spatial software visualization in the IDE: an exploratory study," in *Proc. of the 5th International Symposium on Software Visualization SOFTVIS'10*. ACM, 2010, pp. 113–122.

[9] R. DeLine and K. Rowan, "Code canvas: zooming towards better development environments," in *Proc. of the ACM/IEEE International Conference on Software Engineering ICSE'10*. ACM, 2010, pp. 207–210 vol. 2.

[10] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *Proc. of the ACM/IEEE International Conference on Software Engineering ICSE'10*. ACM, 2010, pp. 455–464 vol. 1.

[11] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," in *Proc. of the Working Conference on Reverse Engineering WCRE'05*. IEEE Computer Society, 2005, pp. 133–142.

[12] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[13] M. Foucault, *The Order of Things: An Archaeology of the Human Sciences (originally published in French: Les mots et les choses - une archologie des sciences humaines)*. Paris: Gallimard, 1966.