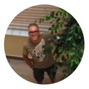
Coherence of comments and method implementations: a dataset and an empirical investigation

Article *in* Software Quality Control · November 2016

DOI: 10.1007/s11219-016-9347-1

Anna Corazza
University of Naples Federico II
69 PUBLICATIONS **437** CITATIONS
SEE PROFILE


Giuseppe Scanniello
Università degli Studi della Basilicata
175 PUBLICATIONS **1,063** CITATIONS
SEE PROFILE

Coherence of Comments and Method Implementations: a Dataset and an Empirical Investigation

Anna Corazza
Dept. of Electrical Engineering
and Information Technologies
University of Naples "Federico II"
Naples, Italy
Email: anna.corazza@unina.it

Valerio Maggio
Fondazione Bruno Kessler
Trento, Italy
Email: vmaggio@fbk.eu

Giuseppe Scanniello
Dept. of Mathematics, Information
Technology, and Economics
University of Basilicata
Potenza, Italy
Email: giuseppe.scanniello@unibas.it

Abstract—**In this paper, we present the results of a manual assessment on the coherence between the comment and the implementation of 3636 methods in 3 open source software applications (for one of these applications, we considered 2 different subsequent versions) implemented in Java. The results of this assessment have been collected in a dataset we made publicly available on the web. The creation of this dataset is based on a protocol that is detailed in this paper. We present that protocol to let researchers evaluate the goodness of our dataset and to ease its future possible extensions. Another contribution of this paper consists in preliminarily investigating on the effectiveness of adopting a Vector Space Model (VSM) with the *tf-idf* schema to discriminate coherent and non-coherent methods. We observed that the lexical similarity alone is not sufficient for this distinction, while encouraging results have been obtained by applying an Support Vector Machine (SVM) classifier on the whole vector space.**

Keywords - ***Comment Coherence; Maintenance; Experimental Protocol; Dataset; Lexical Information; Classification;***

I. Introduction

Commenting source code has long been a common practice in software development. Developers use comments because they are more direct, descriptive, and easy-to-understand than source code statements alone. Comments are important as they are used to convey the main intent behind design decisions, along with some implementation details. Developers also use comments to document source code and to embed their knowledge about the domain of that code in identifier names [6]. In this respect, the information in source code statements and comments represents a valuable resource for developers to maintain and evolve software applications [29]. Unfortunately, comments may convey information unrelated or inconsistent with the source code to which it refers to. This could happen for several reasons. For example, developers could modify the intent of source code while executing a maintenance operation, without updating its comment accordingly [27].

In this paper, we describe the protocol used to collect the data needed to create our dataset on the coherence between the lead comment[1] of methods and their implementations. The focus of our study is to investigate whether the lead comment of a method properly conveys the main intent of the entire method it refers to, along with implementation details (e.g., types of parameters and of returned values). In this scenario, we do not take into consideration in-line comments because they only provide information for very specific parts of the code base. Hence, there exists coherence between a lead comment of a method and its source code (also simply coherence, from here on) if that comment describes the intent of the method and its actual implementation. Our dataset contains annotations on the coherence of $3,636$ methods concerning 4 implementations of 3 different software written in Java and implemented in open source projects. The creation of this dataset is based on a protocol we describe in this paper as well. Accordingly, researchers can evaluate the goodness of our dataset and easily extend it with applications written with different programming languages and from different kinds of projects and application domains. We made our dataset publicly available for the download on the web. This will also allow researchers to assess their approaches to estimate method coherence. In this paper, we also discuss on qualitative results obtained from the dataset creation. Another contribution of our paper consists in preliminarily studying possible links between coherence and lexical information provided in source code and lead comment of methods. In this respect, quantitative results are shown and discussed here. We explore potentialities of using lexical similarity (by exploiting a Vector Space Model, also VSM from here on) and the *tf-idf* representation to assess coherence.

The work presented in this paper is built on that presented in [11] and, with respect to it, we list here how that paper has been enhanced:

1) We extended and improved motivation and related work.
2) We better and deeply described the protocol used to

[1]The comment right before/after of the definition of a method, a class, abstract class and so on.

build our dataset.

3) We better investigated possible relationships between coherence and lexical similarity of code and lead comments of methods.
4) We applied feature reduction to study which information, if any, is more discriminant for comment coherence.
5) We further extended our empirical investigation by applying a Support Vector Machine (SVM) classifier to speculate whether these techniques can automatically evaluate coherence.
6) We improved the discussion of possible threats that could affect the validity of our outcomes.

The paper is structured as follows. In Section II, we discuss on motivation and related work, while we present the protocol used to define our dataset in Section III. In this section, we also provide the definition of method coherence. We report qualitative and quantitative results from the creation of our dataset in Section IV. In this section, we also discuss these results and possible threats that could have affected their validity. In Section V, we show and discuss on outcomes from a study we have conducted to investigate on the link between method implementations and their lead comments. Final remarks conclude the paper.

II. Background

A. Motivation

Researchers have largely studied developers behavior while executing maintenance tasks [13], [21], [30]. For example, LaTozza *et al.* [21] conducted a qualitative study on professional developers from a single company. Data were gathered through surveys and interviews. As for maintenance, the authors observed that developers remained focused on the code itself despite the availability of design documents, so concluding that developers considered documentation inadequate to support the execution of maintenance tasks. Similar conclusions were drawn by DeLine *et al.* [13]. Authors conducted a study on seven professional developers, who were asked to update an unfamiliar implementation of a video game. Also, deSouza *et al.* [12] conducted an empirical study on professional developers. Results suggested that developers found source code and comments the most important artifacts to accomplish maintenance tasks.

Roehm *et al.* [26] performed a qualitative study. The used methodological approach was qualitative. Results suggested that developers put themselves in the role of end-users, namely they perform a sort of dynamic analysis on a subject software. Authors also observed that program comprehension was considered a subtask of other maintenance tasks rather than a task by itself. In addition, no program comprehension tool was used because developers were completely unaware of their existence. That is, there is a gap between program comprehension research and practice. As a consequence, the source code represented the only reliable source of information available for developers to deal with comprehension tasks.

On the basis of the related work discussed before, we can assert that developers relies on inadequate design documentation and tools when comprehending unfamiliar source code [21], [25]. Therefore, the only possible strategy for the developer is the analysis of source code (and the comment if present) and/or its execution [26].

Lexical information present in code base is also relevant in automated and semiautomated techniques proposed in the literature to aid developers in the comprehension and evolution of existing software. Many of these approaches exploit the lexical information provided in source code, comments, and other kinds of software artifacts [1], [9], [10], [14], [20], [24], and use Information Retrieval (IR) techniques [23]. As for source code based artifacts, authors often assume an implicit hypothesis that comments and implementations contain similar lexemes [14], [28]. However, this assumption does not always hold because there could be a mismatch between the vocabulary used in source code and that used in comments [22]. In addition, a lead comment may convey information unrelated or inconsistent with the source code to which it refers to. This issue may be caused by developers that modify source code intent without updating its comment accordingly [27].

Concerns introduced before could affect software maintenance and evolution as follows:

1) Lack of coherence (or non-coherence) between comments and source code affects all those approaches developed in software maintenance and evolution field that rely on text search or text retrieval techniques. These (static) approaches are mostly based on information extracted from: source code without executing it, code comments, and other kinds of textual software artifacts. These kinds of approaches are called lexical. For example, we can speculate that lexical approaches may get worse performances when applied to source code containing non-coherent methods because benefits from the use of an IR technique may be obfuscated from the lack of coherence.
2) As software evolves, comments can easily grow out-of-sync. In other words, a comment may convey information unrelated or inconsistent with the source code to which it refers to [27]. When this happens, comments can confuse and mislead developers and the cost to comprehend, maintain, and test source code increases [32], [33].
3) Lack of coherence may also indicate either a fault in source code (e.g., the body of a method) or a fault in the comment that can mislead the method callers so introducing faults in their code [33].

According to what discussed before, it seems advisable to have approaches to assess coherence between implementation and comment and between method source code and its lead comment, in particular. However, at the present time no approaches have been yet proposed in this regard. We can speculate that this is due to a lack of datasets and benchmarks on which research solutions can be assessed. Public datasets and benchmarks would increase confidence on researchers' solutions, so also speeding-up the transfer of these solutions to the industry. The lack of a public dataset could be also due to the effort and complexity related to its creation. This is why we made publicly available our dataset on the web.[2]

B. Related Work

How developers add comments or modify them when the software evolve has been scarcely investigated in the past. For example, Fluri *et al.* [15] investigated whether source code and associated comments are really changed together along the evolutionary history of a software. To this end, three open source software (i.e., ArgoUML, Azureus, and JDT core) were studied. The most salient results can be summarized as follows: *(i)* newly added code barely gets commented; *(ii)* class and method declarations are commented most frequently; and *(iii)* 97% of comment changes are done in the same revision as the associated source code change. A similar study has been conducted by Jiang and Hassan [18]. In particular, these authors analyzed the evolution of comments over time to investigate the common claim that developers change code without updating its associated comments which is likely to cause bugs. In general, authors conclude that identifiers can be useful if carefully chosen to reflect the semantics and role of the named entities. More specifically, results reveal that over time the percentage of commented code remains constant except for early fluctuation. The authors speculate that this result might be due to the commenting style of a particular developer who is more active than others.

Steidl *et al.* [31] presented an approach for quality analysis and assessment of code comments. A model for comment quality based on different comment categories was defined. To categorize comments, the authors proposed a machine learning technique, which was applied on software implemented in Java and C++. Quality attributes of comments were assessed by two metrics. The first metric estimated coherence between code (i.e., method name) and comment of methods, while the second considered the length of (inline) comments. As for coherence, authors extracted words contained in the comment and compared it to the words contained in the method name. To this end the Levenshtein string edit distance was exploited. The rationale behind the use of the length of comments is that shorter inline comments contain less information than longer ones. The authors assess their proposal by means of a survey among experienced developers. There are several differences between the work by Steidl *et al.* [31] and ours. The most remarkable difference is that we do not propose a method for estimating comment quality, but a public dataset to be used, for example, to assess the performance of approaches able to infer coherence. In addition, we present a preliminary investigation to study the effectiveness of adopting a VSM with the *tf-idf* schema to discriminate coherent and non-coherent methods. Finally, we apply an SVM classifier to speculate whether these techniques can automatically evaluate comment coherence.

[2]www2.unibas.it/gscanniello/coherence/

III. Research Design

The method we used to create our dataset is inspired by the perspective-based and checklist-based review methods [36]. Several perspectives can be used while applying a perspective-based approach. In our case, the perspective is that of the Researcher, who aims to assess coherence between lead comment of methods and their implementation. Reviews are based on a set of items we derived in brainstorming and working meetings conducted either face-to-face or remotely.

In this section, we present the fundamentals of the research presented in this paper. In particular, we describe how we defined the protocol to create our dataset and how we applied it on a number of software applications included in that dataset. The creation of our dataset allowed us to observe some patterns in the gathered data and to distill them in outcomes useful for the researcher. These outcomes are both qualitative and quantitative. Our main intent is twofold: we want to support researchers to evaluate the quality of the provided data, and we aim to aid them in extending the dataset by including new software applications. This latter goal can be pursued by applying our process to collect data and to prepare them to extend our dataset with new applications.

We believe that the creation of a dataset has to be founded on clear and well defined tasks and subprocesses. In this respect, we can speculate that the effect of this design choice is to have a resulting dataset as authoritative as possible. Therefore, our research is founded on:

1) **Working Meetings.** We discussed to determine the goals of our work, along with the process used in the creation of dataset. Meetings occurred face-to-face or virtually, as mediated by communications technology.
2) **Dataset Creation.** We created the dataset according to our defined process. People other than the authors were involved.
3) **Outcomes.** We gathered several information and data during and after the creation of our dataset. Information and data were synthesized in outcomes.
4) **Publishing Results and Dataset.** We shared our experience and outcomes with the research community and released our dataset on the web.

A. Working meetings

We organized a series of 7 working meetings, each taking between 1 and 2.5 hours, conducted face-to-face or virtually via Skype. Results from each meeting were documented to share outcomes and the rationale behind decisions among the authors and people involved in the creation of our dataset (i.e., annotators,[3] from here on). In Table I, we sketched the purposes and the outcomes of occurred meetings.

In the first meeting (i.e., meeting #1), we discussed on a series of research opportunities concerned with commenting source code. We used brainstorming to generate research ideas on this topic. On the basis of motivations given in

[3]In our case, an annotator is a person that produces annotations to software associating coherence information to methods.

TABLE I: Meeting Schedule

Meeting	Purpose	Outcome
#1	Determine preliminaries.	A series of goals and possible protocols to define our dataset were discussed. At the end of the meeting, we settled the protocol and the goals highlighted in this paper.
#2	Specify coherence meaning.	We specified the meaning of coherence between the lead comment of a method and its implementation.
#3	Create the process.	We defined the process to create the dataset, along with the checklist to be used by annotators during the process.
#4	Review process and checklist.	Both the process and the checklist were reviewed. Issues were solved and the final version of each of these artifacts was produced.
#5	Establish assessment protocol and evaluation criteria.	We determined the assessment protocol to apply during the evaluation of method coherence. Moreover, we established few quantitative and qualitative criteria to analyze the dataset.
#6	Distill outcomes.	We reported results obtained from the creation of the dataset.
#7	Plan next steps of research.	A future research plan was established.

Section II-A, we decided to focus on coherence. At the end of meeting #1, we settled the protocol and the goals highlighted in this paper. In meeting #2, we focused on what coherence between lead comments and method implementations means in our research. Further details on this matter are reported in Section III-B. In meeting #3, we discussed and defined the process to create our dataset (see Section III-C, for details). We exploited brainstorming to generate ideas also in meeting #3. Since one of the most important outcomes for this part of our research was the definition of a dataset, we planned another meeting (meeting #4), where we reviewed the process and artifacts produced (e.g., checklist to verify method coherence). The goal was to solve possible issues (e.g., check items to be modified to avoid confusion) and produce the final version of the artifacts needed for the dataset creation. In meeting #5, we defined the assessment protocol to evaluate method coherence, and the set of evaluation criteria adopted to analyze data gathered from the creation of the dataset. In Section III-D, we give details on purpose and outcomes of meeting #5. Having built our dataset, we had a meeting (meeting #6) to discuss and distill outcomes from gathered data (Section IV). The nature of outcomes was both quantitative and qualitative (see Section IV-A and Section IV-B, respectively). In meeting #7, we discussed on possible directions for our research. Results concerned to one of these directions are reported in Section V. This part of our research clearly represents a new contribution with respect to our previous paper [11]. Other possible research directions from meeting #7 are sketched in Section VI. It is worth mentioning that we used brainstorming in all those cases in which we had to deal with complex problems and we need creative solutions for these problems. The face-to-face meetings were: meeting #1 and meeting #3. Other meetings were conducted remotely via Skype.

B. Method Coherence

A comment is a construct of a programming language to embed natural language annotations in source code. Comments are supposed to be significant for developers and maintainers, and not for compilers and interpreters. There are many tools and services (e.g., Sphinx,[4] Javadoc, and readthedocs.org) originally developed to automatically generate software documentation from source code comments. Accordingly, developers should use comment construct to make software properly documented and to improve source code comprehensibility and maintainability. However, flexibility provided by comments (that represents a strength point for this kind of construct) often results in a set of not really helpful and useful information for those developers who did not originally authored source code.

[4]http://sphinx-doc.org

To make source code comments helpful, there are divergent points of views on how they should be written [19]. Developers suggest to use comments as a means to outline design intentions and goals (i.e., the *what*). Therefore, source code has to provide details about the *how* and it should include identifiers whose names are short, meaningful, and mnemonic [27]. Others suggest to use comments to provide additional insights behind implementation decisions, thus focusing more on what source code actually implements rather than its main intent. We might imagine that each strategy to comment source code has advances and disadvantages. Surprisingly, there is a lack of a solid body of knowledge on this topic.

At the end of meeting #2, we realized that the text of a lead method comment should properly describe concepts (i.e., design goals and/or implementation details) which can find correspondences in the specific implementation of that method. Moreover, if the lead comment provides details about input parameter/s, its/their intended use should be properly described. In case all these conditions hold, the lead comment of a method can be considered *coherent* with its implementation. In this respect, it is important to emphasize that our definition of coherence does not contemplate methods with no implementation (i.e., interface or abstract methods). Even if these methods could have some commenting per se, annotators did not work on methods with no body.

In Figure 1, we report the implementation and the lead comment of `setTickLabelsVisible` (extracted from JFreeChart ver. 0.7.1, included in our dataset). According to our definition of coherence, we can assert that this method is coherent. On the other hand, Figure 2 and Figure 3 provide two different examples of non-coherence, reporting methods gathered from JFreeChart ver.0.6.0 and JHotDraw respectively, also included in our dataset. Figure 2 reports three methods extracted from three different classes, but sharing the same lead comment. This comment does not reflect any details of the corresponding implementations (see Figures 2a, 2b, and 2c).

```
/**
 * Sets the flag that determines whether or not
 * the tick labels are visible.
 * Registered listeners are notified of a
 * general change to the axis.
 *
 * @param flag The flag to set.
 */
public void setTickLabelsVisible(boolean flag) {
    if (flag!=tickLabelsVisible) {
        tickLabelsVisible = flag;
        notifyListeners(new AxisChangeEvent(this));}
}
```

Fig. 1: The lead comment and the implementation of the method `setTickLabelsVisible` of JFreeChart 0.7.1

```
/**
 * Returns the number of series in the data source.
 * @return The number of series in the data source.
*/
public int getSeriesCount() {
    return 0;
}
```

(a) getSeriesCount@EmptyXYDataset

```
/**
 * Returns the number of series in the data source.
 * @return The number of series in the data source.
*/
public int getSeriesCount() {
    return 1;
}
```

(b) getSeriesCount@SampleXYDataset

```
/**
 * Returns the number of series in the data source.
 * @return The number of series in the data source.
*/
public int getSeriesCount() {
    return 2;
}
```

(c) getSeriesCount@SampleHighLowDataset

Fig. 2: Non-Coherent methods in JFreeChart 0.6.0

Conversely, the `save` method in Figure 3 provides a very poor and inadequate description of the design intent of the method, thus reflecting a lack of coherence with the underlying implementation.

C. The Process

The process to create our dataset is composed by two main consecutive steps:

- **Verify coherence**. To establish if there is coherence or not, multiple annotators have to be involved. Annotators individually verify the presence of coherence between the lead comment of a set of methods and their corresponding implementations. To this end, annotators use the checklist we have defined according to our definition of method coherence. This checklist is reported in Table II. The items of this checklist aim to verify the existence of several aspects concerned to method coherence. It is worth mentioning that this checklist only applies if the lead comment of a method is present.

```
// GEN-FIRST:event_save
// Code for dispatching events from components
// to event handlers.
private void save(java.awt.event.ActionEvent evt) {
  try {
        String methodName = getParameter("datawrite");
        if (methodName.indexOf('(') > 0) {
                methodName = methodName.substring(0,
                        methodName.indexOf('(') - 1); }
        JSObject win = JSObject.getWindow(this);
        Object result = win.call(methodName, new
                Object[]{getData()});
  } catch (Throwable t) {
        TextFigure tf = new TextFigure("Fehler:␣" + t);
        AffineTransform tx = new AffineTransform();
        tx.translate(10, 20);
        tf.transform(tx);
        getDrawing().add(tf); }
}
```

Fig. 3: Non-Coherent method in JHotDraw 7.4.1

TABLE II: Used checklist

ID	Check item
1	The lead comment of the method describes the intent of the source code of this method.
2	The intent described in the lead comment of the method corresponds to the actual implementation of this method.
3	The lead comment of the method describes all the expected behaviors of the actual implementation of this method.
4	If the lead comment of a method provides implementation details (e.g., names and types of input parameters according to JavaDoc), this information is aligned with the implementation of this method.
5	If the lead comment of the method provides details about input parameters, their intended use is properly described.

- **Resolve conflicts**. In pair sessions, experts (second and third authors of this paper) analyzed methods for which annotators' coherence judgement was different. Experts were asked to analyze methods (i.e., implementations and comments) on which annotators did not reach a consensus. That is, the application of our checklist led to different decisions. Please note that deciding if a method is coherent or not is however based on human judgments even if a checklist is applied. Methods on which experts do not get an agreement have not to be included in the dataset (experts always got an agreement in our case). To resolve conflicts, we recommend the use of a reading technique based on our checklist. The agreement among annotators has been estimated by applying the *kappa index* [7].

To support the execution of the two steps described just before, we implemented a web application. Further details on this application are not given for scant relevance.

D. Assessment Protocol and Evaluation Criteria

Each annotator analyzed in isolation the lead comment and the source code of each assigned method. He/she had to look only at source code and comment. To this end, annotators used our supporting web application. In case annotators would need additional information to make their decisions on a given method, they should be able to access source code files of software where this method was extracted from. Our web application provided support also for this task. For each method, annotators had to report their judgement by selecting one of the following three values: *Non-Coherent*, *Don't Know*, and *Coherent*. The former and the latter judgment correspond to negative and positive evaluations, respectively. The second judgment (*Don't Know*) indicates a neutral response. We asked annotators to use a neutral response when it was very difficult (or not possible) to take a decision on method coherence. In other words, the application of our checklist did not allow annotators to take a decision. Annotators did not know one another.

Our protocol relies on the evaluations of multiple annotators. Multiple annotators allow us to distribute the effort required to perform evaluations. This strategy also reduces possible evaluation biases that are intrinsically induced by involving single annotators. We asked each annotator to evaluate a subset of methods in the whole dataset. We imposed only one constraint in our annotation process: each method had to be evaluated by two annotators at least. This approach is widely used in the literature (e.g., [3]).

In this scenario, it is possible to calculate agreement among annotators, as an estimation of the reliability of their evaluations. If annotators' agreement degree is high on all the methods, then the annotation can be considered reliable. As mentioned before, we used the *kappa index* [7] to estimate annotators' agreement and then annotation reliability. We opted for this index because widely used for purposes similar to ours. In particular, it is designed for categorical judgments and refers the agreement rate calculation to the rate of chance agreement:

$$k = \frac{p_o - p_c}{1 - p_c}, \tag{1}$$

p_o is the observed probability of agreement, while p_c is the probability of agreement by chance. By a simple algebraic manipulation, Equation 1 can be written as:

$$k = 1 - \frac{q_o}{q_c}, \tag{2}$$

where $q_o = 1 - p_o$ and $q_c = 1 - p_c$. Indeed, q_o and q_c are the observed and the chance probabilities of *disagreement*, respectively. The index assumes values in $]0, 1]$ when the observed disagreement is less likely than chance. This index assumes a null value if it is exactly as likely as chance, and a negative value when it is more likely than chance probability of disagreement. Perfect agreement corresponds to $k = 1$. Values greater than 0.80 are usually considered as a cue of good agreement. Values in the interval $[0.67, 0.80]$ are considered acceptable [7].

The classical formulation of kappa index considers a binary classification problem (e.g., *Non-Coherent* or *Coherent*). However in our case, the neutral judgement (i.e., *Don't know*) is also considered. Therefore, possible disagreements also include the case where one of the two answers is the neutral one. In this case, it is possible to differently weigh the possible disagreements among annotators. In fact, disagreements due to the neutral answers are less serious than disagreements where judgments are totally divergent (i.e., *Coherent* and *Non-Coherent*, in our case). To this end, Cohen [8] presents a variant of the kappa index, where in case of a disagreement, different weights can be introduced to evaluate its seriousness. In case the same weight is assigned to all possible disagreement combinations, the original (unweighed) formulation is obtained.

TABLE III: Descriptive statistics.

Application	Version	Files	Classes	Methods	Methods with Comments
CoffeeMaker	-	7	7	51	47 (92%)
JFreeChart 6.0	0.6.0	82	83	617	485 (79%)
JFreeChart 7.1	0.7.1	124	127	807	624 (77%)
JHotDraw	7.4.1	575	692	6414	2480 (39%)

The formulation of the *Weighed Kappa* (WK) is:

$$k = 1 - \frac{W \times O}{W \times C}, \tag{3}$$

where O is the matrix of observed scores, C is the matrix of expected scores based on chance agreement, and W is the weight matrix. Equation 3 can be seen as equation 2 if for the computation of q_o and q_c the contributions are weighed according to the importance given to the corresponding disagreement cases. We assign to the *Don't know* response a weight that is half the weight assigned to the *Non-Coherent* (or *Coherent*) response. The used schema is the same as that proposed by Cohen [8].

IV. Our Dataset

The creation process of our dataset lasted from January, 15^{th} 2014, to June, 20^{th} 2014. A total of 800 man/hours were needed for its creation. The number of man/hours gives an estimation on the effort required to conduct a part of the study presented in this paper. This information also provides an indication on the effort and the time the interested researcher needs to extend our dataset.

In Table III, we report some descriptive statistics automatically gathered from the software included in our dataset. Three out of four applications have been implemented in open source projects. All the four applications in our dataset were written in Java. Details on these applications follows:

- **CoffeeMaker**[5] is an application to manage inventory and recipes and to purchase beverage. We chose this coffee maker management application because it is well designed and implemented (being developed for educational purposes). The percentage of methods with a lead comment was 92%. This indicates that developers made an extensive use of comments.
- **JFreeChart**[6] is a Java tool supporting visualization of data charts (e.g., scatter plots and histograms). We included two versions of this application in our dataset.

[5]agile.csc.ncsu.edu/SEMaterials/tutorials/coffee_maker/
[6]www.jfree.org/jfreechart/

As shown in Table III, almost 80% of the methods in both considered versions of this application have lead comments. This suggests that developers made an extensive use of comments. This is one of the reasons to have chosen JFreeChart.

- **JHotDraw**[7] is a framework for technical and structured graphics. Even if the source code of JHotDraw is scarcely commented (see Table III), it is well-known in the software maintenance field due to its good implementation. These reasons suggested to include JHotDraw in our dataset. Moreover we believed that it was important to choose an application with less lead comments than others. This could possibly allow us to find differences in the coherence of comments related to differences in percentage of methods with lead comments.

Three annotators took part in the annotation process of our dataset. Two of them have similar background and experience. In particular, they hold a Bachelor degree in Computer Science from the University of Basilicata and their curricula were similar. The third annotator can be considered more experienced than the other two. He had a Master degree in Computer Science. In particular, he got the Bachelor degree from the University of Rome "La Sapienza", while the Master Degree has been from the University of Basilicata. We randomly assigned annotators to the selected software applications. However, we tried to balance the effort among annotators as much as possible. We also guaranteed that each method was evaluated at least by two different annotators. Moreover, we asked each annotator to double check their judgements before considering them fully committed. This was what we established in the protocol defined to create our dataset.

A. *Quantitative Results*

In Table IV, we report both the values of the Unweighed Kappa formulation (i.e. UK), and the Weighed Kappa formulation (i.e. WK) obtained at the end of the first step (i.e., *Verify coherence*) of the process described in Section III-C. As shown in this table, the agreement among annotators is good on CoffeeMaker and JFreeChart (both versions). For these applications, values for both WK and UK are greater than 0.9. As for JHotDraw, the agreement seems acceptable (values for UK and WK are equal to 0.824 and 0.684, respectively). However, the difference between the values for UK and WK is large, thus indicating that the WK formulation provides a more accurate indication on the agreement between the evaluations performed on this application. As compared with other applications, we observed that in more cases annotators disagreed on the coherence between lead comment and method implementation of JHotDraw. This justifies why values for UK and WK are inferior to those for other applications. We also noted that totally divergent judgments (i.e., Coherent and Non-Coherent) were more common for JHotDraw. This outcome explains the differences in the values for UK and WK.

[7]www.jhotdraw.org/

TABLE IV: Agreement Rate of Judges as computed by the Cohen's Kappa Index. **UK**: Unweighed Kappa Index; **WK**: Weighed Kappa Index

Application	**UK**	**WK**
CoffeeMaker	0.913	0.913
JFreeChart 6.0	0.939	0.918
JFreeChart 7.1	0.983	0.977
JHotDraw	0.824	0.684

```
/**
* Returns the number of items in the specified series.
* @param series The index (zero-based) of the series;
* @return The number of items in the specified series.
*/
public int getItemCount(int series) {
    return 47;
}
```

(a) getItemCount@SampleHighLowDataset in JFreeChart 0.7.1

```
/**
 * Writes the view to the specified uri.
 * @param f The uri/file where the view is written
 * @param chooser
*/
@Override
public void write(URI f, URIChooser chooser)
throws IOException {
    Drawing drawing = view.getDrawing();
    OutputFormat outputFormat =
        drawing.getOutputFormats().get(0);
    outputFormat.write(new File(f), drawing);
}
```

(b) write@PertView in JHotDraw 7.4.1

Fig. 4: Example of two methods reviewed during the *Resolve Conflicts* step.

Once *Verify coherence* was accomplished, the total number of methods on which annotators did not agree was 302, corresponding to 8.3% of the total number of methods in all the applications in our dataset. As UK and WK values estimated, most of these methods are those in JHotDraw. The second and the third authors of this paper reviewed all these methods in the second step of our process (i.e., *Resolve conflicts*). In Figure 4 we report an example of two methods reviewed in this step. These methods are extracted from JFreeChart 0.7.1 and JHotDraw 7.4.1, respectively. We analyzed the source code and the comment for these methods and concluded that both of them are non-coherent. As a matter of fact, the former violates the third item in our checklist, i.e., the lead comment does not describe all the intended behavior of the actual implementation. Even if the general intent of the method is reported, the comment lacks to mention that the value of 47 is consistently returned. On the other hand, the lead comment of the latter method is not compliant with the last item of the checklist. In fact, no description of the intended use for the `chooser` parameter is reported, likely because this is never used in the code. The two authors of this paper reached an agreement on all the 302 methods, which have been finally included in the dataset. In Table V, we summarize the total number of methods in the final version of our dataset.

TABLE V: Descriptive Statistics of the Dataset [a]

Application	C	NC	Total	Don't Know (Not included)
CoffeeMaker	27	20	**47**	0
JFreeChart 6.0	406	55	**461**	24
JFreeChart 7.1	520	68	**588**	36
JHotDraw	762	1025	**1787**	693
Total	**1715**	**1168**	**2883**	

[a]C stands for Coherent, NC for Non-Coherent

To further investigate the impact of including two versions of the same software in the dataset, namely JFreeChart 0.6.0 and 0.7.1, we analyzed the methods contained in both the applications and grouped them in terms of differences (and/or similarities) in their code and comment. The resulting groups are sketched in Figure 5. In more details, the two versions of JFreeChart included in the dataset have a total of 283 methods in common. Most of these methods ($\approx 91\%$) have no differences in the lead comments, apart from small variations in the layout and in the formatting of the text. Moreover, most of these methods also share the same implementation (i.e., 225 methods out of 257). Only 32 of these methods have differences in the corresponding code. On the other hand, among the 26 methods having different lead comments, 10 have also different implementations, while 16 have no variations in the code. According to these four groups of methods, we investigated how the coherence changes in the two versions of the application. As expected, methods sharing the same code and comment have no difference in terms of coherence. This provides additional evidence of the quality of the created dataset. Similarly, no difference in coherence has been found between methods with changes only in the implementation. Further manual investigations on these methods confirmed that the differences in the code were limited to small refactoring operations (e.g., variable renaming or `try-catch` blocks additions), which did not affect the semantic of the whole implementation. Conversely, the analysis of coherence of methods with different lead comments provided insights that are worth discussing. As for the methods sharing the same implementation, most of the changes in the comments are related to improvements in the Javadoc syntax (e.g., addition of `@param` and `@return` annotations), along with revisions in description of the intent of the method. In facts, these variations do not affect the corresponding coherence, as 14 (out of 16) methods in this group have the same coherence in the two versions of the application. The other 2 methods present only partial additions in the comments that were not actually aligned with the code. Consequently these methods were correctly evaluated as non-coherent. Finally, among the 10 methods with different code and comments, 9 of them have the same coherence. This is the case in which code and comments have been correctly updated during the evolution of JFreeChart. The only difference in coherence found in the methods of this group corresponds to a case in which a non-coherent method (in the 0.6.0 version) becomes coherent

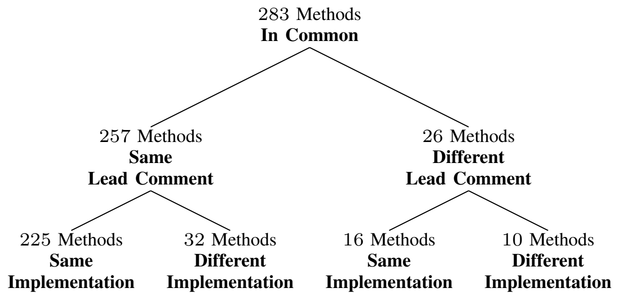


Fig. 5: Relationships in terms of differences and similarities in code and comment for methods included in the two versions of JFreeChart (0.6.0 and 0.7.1)

after improvements in code and comment. Further details on the analysis performed on the methods included in the two versions of JFreeChart are available online,[8] and included in the replication package.

B. Lessons Learned from Dataset Creation

During the dataset creation process, we obtained some qualitative lessons. Some of these lessons come up from our interaction with annotators, while others from a post-analysis we performed on our dataset as well as from our direct effort in its creation process (i.e., *Resolve Conflicts*). Our lessons learnt can be summarized as follows:

- **Duplicated Comments.** Applications often contained methods with duplicated comments. This often happened for those methods implementing interface APIs. Comments were likely generated by IDEs (Integrated Development Environments) used in the development process. As a consequence, duplicated comments did not provide meaningful information on different implementations of methods. This scenario is particularly common for JHotDraw.
- **Delegation Methods.** A "special" case of duplicated comments refers to delegation methods, namely methods that contain only a single method invocation in their body. For example, this happens in the methods of a "Middle Man" bad smell class [16]. We noticed that the lead comment of these methods is exactly the same of the delegated method and such a delegation is not commented in the source code. According to our definition of coherence, delegation should be commented.
- **Propagation of Comments through Class Hierarchies.** Overloading and overriding methods do not contain any lead comment documenting their specific behavior or intent. This happens for both intra-class and inter-class methods. Figure 2 shows an example of this case: three methods (Figures 2a, 2b, and 2c) extracted from three different classes sharing the same super abstract class.[9] For all the three methods, the comment is exactly the

[8]https://goo.gl/5oEys8

[9]http://www.jfree.org/jfreechart/api/javadoc/org/jfree/data/general/AbstractSeriesDataset.html#getSeriesCount

same, not reflecting the difference in the corresponding implementation.

- **Method Signatures.** This is one of the most common cases leading to non-coherence: documented method signature did not have correspondence with actual implementation (e.g., there is not correspondence between types and names of input parameters). That is, comments are very often not updated during software evolution. This is a very common (bad) practice [18] also for professional developers [27]. We recognized this issue in all the applications in our dataset.
- **Comments describing Method Usage.** Lead comment describes the usage of the method implementation (i.e., how and when the method is invoked) instead of its intent and behavior. This usually affected some overloaded methods and event handlers in JHotDraw. In the cases before, developers mostly do not update comments as consequence of either automatically generated or modified source code. On the other hand, in this scenario the use of comment construct does not fit with our definition of coherence. We observed that this was not so common in the considered applications.
- **Annotator Conflicts.** The agreement among annotators was generally good. We observed conflicts only for 8.3% of the total number of methods in all the applications in our dataset. However, we observed a larger number of conflicts on the largest application, namely JHotDraw. This may suggest that the larger the application, the worse for a developer to assess coherence is. We also observed that in a few cases annotators had totally divergent judgments on methods and lead comments (see Section IV-A). We cannot provide a plausible justification for this result. Deciding if a method is coherent or not is based on human judgments although in our study it is guided by a checklist. As future work, it could be interesting to investigate on the decision making process to establish when a method implementation is coherent with its leading comment.

Our qualitative lessons suggest that most of the critical aspects related to lack of coherence is due to duplication of comments introduced by either developers or IDEs. These comments are usually not modified to specify the intent of the code to which they refer to. Furthermore, developers also tend to modify the intent or the behavior of the source code, without updating the comment accordingly.

C. Threats to Validity

To better comprehend strengths and limitations related to our dataset, the threats that could affect the validity of attained results are presented and discussed. Despite our efforts to mitigate as many threats to validity as possible, some are unavoidable.

Threats to the *conclusion validity* concern the actual coherence between lead comment of methods and their implementation. To deal with this threat the kappa index has been used. This index provides an indication on the reliability of evaluations.

Internal validity threats are related to annotators. They had different levels of expertise. To force them to perform their coherence evaluations individually, the annotators did not know each other. Some additional threats to internal validity regard the evaluation process and the defined checklist.

Possible threats to *construct validity* might concern the protocol to collect data and to create our dataset and social factors. Although we used a checklist-based review method, coherence relies on human judgment, which is a potentially error-prone process. To deal with construct validity, annotators did not know the final goal of our study, and they were not involved in either any discussions nor working meetings.

The most important threats to *external validity* are related to the software applications considered in our dataset. For example, all the applications were implemented in Java and this could affect results. For example, Java is more verbose than other programming languages (e.g., C) and then the original developers of the applications in our dataset could have paid scant attention on commenting methods. Another threat to external validity is related to the number of applications in the dataset (i.e., the sample size) as well as the representativeness of these applications. To deal with the sample size, we detailed the protocol used to create our dataset to easily allow its extension. Regarding the second kind of threat, we selected software applications from different domains. Although our effort to deal with the representativeness of these applications, two of them have been designed as educational applications (i.e., JHotDraw and CoffeMaker). In other words, it could be possible that these applications have been developed posing different focus on aspects related to: implementation, evolution, and source code commenting. In addition, JHotDraw and JFreeChart have similar application domains. Another possible threat to external validity concerns the percentage of methods without comments in JHotDraw (i.e., 69%). This aspect could be interpreted in several different ways, for example, developers were not accustomed to use comment construct at all or they exploited this construct only when strictly necessary. On the basis of our results and the available information, we cannot make any plausible justification. All the applications, except CoffeMaker, were open-source and were developed using the same coding conventions (i.e., camel case). We believe that the use of coding conventions should slightly affect external validity whatever is the kind of application (commercial vs. open-source). For example, the use of camel and underscore case conventions should not impact on the meaningfulness of identifier names, while it could affect the capability of developers to recover the lexical information from the source code. As for the kind of application, it could possible that in the software industry developers pay more attention to the internal quality of source code (e.g., code reviews and software inspection could be used). This clearly impacts on our results. Therefore, the study of commercial software represents a possible direction for our future work.

V. On the Relation between Method Implementation and Comment

In this section, we describe the study we conducted on the coherence and the lexical information contained in the source code of methods. As a first step in our investigation, we represent the terms in the lead comment and the implementation of methods of our dataset in a Vector Space Model (VSM) by means of the *tf-idf* schema [23]. In this regard, a special processing pipeline has been applied to extract the lexical information from the source code. In particular, the LINSEN algorithm [9] has been used to properly cope with identifiers composed by multiple words and abbreviations. Moreover, we processed the text of the lead comment and of the implementation separately. In our case, the implementation of a method includes all the terms gathered from both the signature and the body (i.e., code statements and in-line comments).

The *tf-idf* score is defined as the product between the *term-frequency* (*tf*), and the *inverse document frequency* (*idf*). The former considers the number of occurrences of a term in a document (i.e., c), while the latter evaluates the rareness of a term in the collection. That is, *tf-idf* score expresses the relevance of a term with respect to a given document in the collection. In our context, the term "document" refers to either the implementation or a comment of a method, whilst "collection" indicates the whole set of documents, namely all the comments and the implementations of methods in a single software. More specifically, we used the following two formulations: $tf = 1 + \log(c)$, and $idf = \log\left(\frac{N}{df}\right)$, where N is the total number of documents in the collection, while df indicates the number of documents in the collection containing a given term.

The adopted representation of documents in the VSM corresponds to considering a *bag-of-words* model, where only the lexical content is taken into account, while disregarding word order, and other syntactical information. Although this could be seen as a restriction of the approach, this simplification has widely shown its potentialities in information retrieval, text categorization, and many other application domains (e.g., [23]).

A. Lexical Similarity

We might postulate that there is coherence between the comment and the implementation of a method when they have a high lexical similarity. A widely used technique to compute lexical similarity between two documents in the VSM is based on the *cosine similarity*:

$$sim(\mathbf{lc}, \mathbf{sc}) = \frac{\mathbf{lc} \cdot \mathbf{sc}}{||\ \mathbf{lc}\ ||\ ||\ \mathbf{sc}\ ||} \tag{4}$$

where **lc** and **sc** refer to the vectors representing the lead comment and the implementation, respectively. In the literature, these vectors are usually referred as *feature vectors* [23]. Each element in the vector corresponds to a single term extracted from the collection, whose value corresponds to the *tf-idf* score. The general idea is: the more a term is relevant to both the compared documents, the greater its contribution to the cosine similarity is.

TABLE VI: Lexical Similarities for *Coherent* methods.

Application	Methods	**Min**	**Max**	**Med.**	μ_c	σ_c
CoffeeMaker	27	0.16	0.77	0.43	0.51	0.17
JFreeChart 6.0	406	0.0	0.95	0.42	0.42	0.20
JFreeChart 7.1	520	0.0	0.96	0.46	0.46	0.20
JHotDraw	762	0.0	0.97	0.46	0.47	0.22

TABLE VII: Lexical Similarities for *Non-Coherent* methods.

Application	Methods	**Min**	**Max**	**Med.**	μ_n	σ_n
CoffeeMaker	20	0.16	0.82	0.54	0.47	0.18
JFreeChart 6.0	55	0.0	0.73	0.30	0.32	0.18
JFreeChart 7.1	68	0.0	0.81	0.34	0.34	0.22
JHotDraw	1025	0.0	0.93	0.32	0.30	0.23

For the sake of precision, let us underline that cosine similarity is computed by only comparing the documents corresponding to lead comment and implementation referring to the same method. Furthermore, we limited our analysis to the methods that have been labeled as either *Coherent* or *Non-Coherent* at the end of the dataset creation phase. Statistics for these two cases are reported in Table VI and Table VII, respectively. For each application, these tables report the minimum, the maximum, the median, and the mean (μ_c and μ_n) of the cosine similarity values, along with corresponding standard deviation values (σ_c and σ_n). Histograms of the similarity measures are reported in Figure 6. It can be observed that the mean of the lexical similarity is lower in the case of methods labeled as *Non-Coherent*. As for CoffeeMaker, the difference in the lexical similarity between coherent and non-coherent methods is even lower (see Tables VI and VII, and Figure 6). The mean of the lexical similarity values is still higher for methods labeled as coherent. Summarizing, the lexical similarity between comments and implementations is quite low in both coherent and non-coherent cases. We can then assert that one of the most important take-away lessons from our outcomes is: *the lexical similarity between pairs of comment and method implementation is low both in case of coherent and non-coherent methods even if the lexical similarity between lead comment and method implementation is slightly greater in case of coherent methods.*

To corroborate this evidence, we apply a statistical test to verify whether the similarity values corresponding to coherent methods have been drawn from a different distribution with respect to those for non-coherent methods. As no normality hypothesis can be made in this case, we applied the *sum of rank of data* test, also known as Wilcoxon sum of rank of Mann Whitney test [17]. Indeed, given the p-values reported in Table VIII, we obtained that the similarity values correspond to two different distributions in all cases except for CoffeeMaker.

This result suggests that the distinction between coherent and non-coherent methods could be simply based on lexical similarity if a proper threshold on the similarity value is

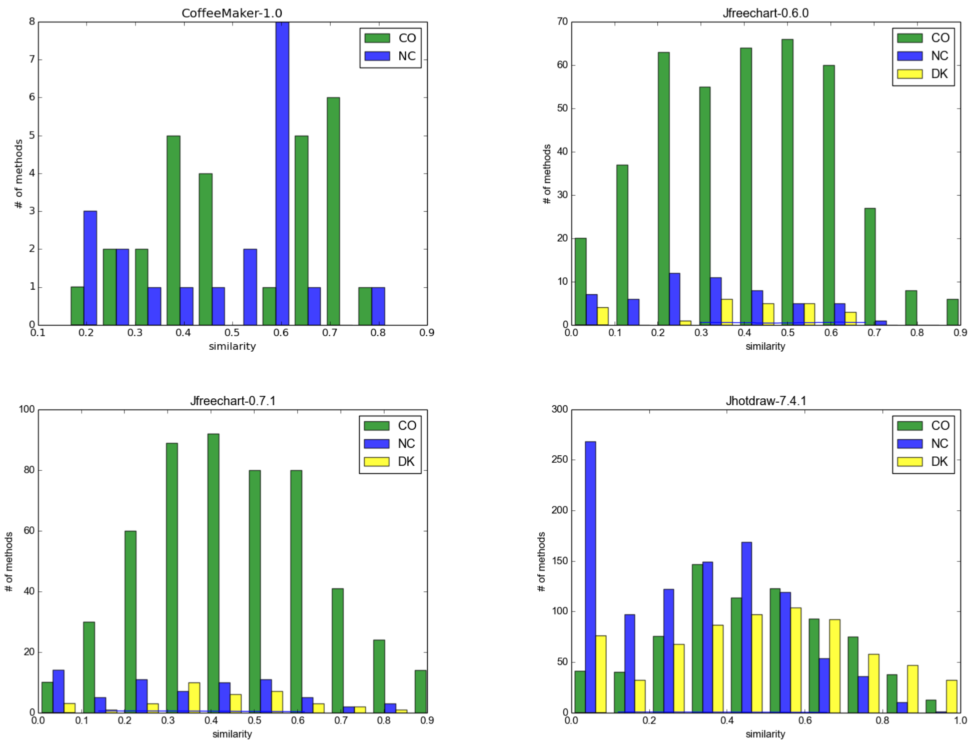


Fig. 6: Histograms of the similarity values for the three categories of methods in the different applications.

TABLE VIII: p values obtained by the sum of rank of data test and the maximum value of F_1 which can be obtained by imposing a threshold on the similarity measure. The last two columns report the F_1 for non-coherent and coherent methods respectively.

Application	**p-value**	**Max F_1**	F_1 NCO	F_1 CO
CoffeeMaker-1.0	0.282004	63.23%	67.92%	58.54%
JFreeChart-0.6.0	0.000773	55.72%	26.14%	85.31%
JFreeChart-0.7.1	0.000012	63.71%	33.96%	93.46%
JHotDraw-7.4.1	0.000000	62.16%	70.83%	53.49%

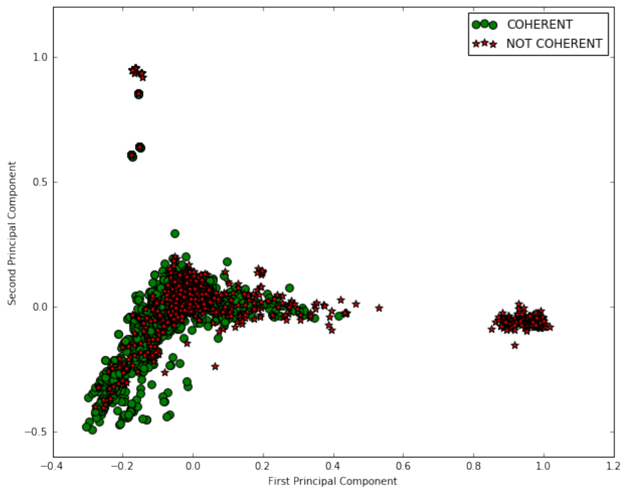
chosen. Therefore, all method-comment pairs whose similarity is lower than this threshold could be labelled as non-coherent. Performance of such a classification could be assessed by accuracy, computed as the rate between the number of correct classifications and the total number of methods classified. However, accuracy is not apt to evaluate classification when the two classes are strongly unbalanced, as for both the versions of JFreeChart , where the number of non-coherent methods is an order of magnitude lower than the number of coherent ones. As a matter of fact, in these cases, a trivial classifier labelling all the methods as coherent indistinctly, would obtain a quite large accuracy (i.e., 86.45% and 86.92%, respectively).

To this end, we adopted the F_1 metric, which can be effectively used even when the number of samples in each class (i.e., coherent and non-coherent) is very different. This metric is defined as the combination of precision (P), and recall (R). In more details, if tp, fp and fn respectively indicate the number of true positives, false positives and false negatives [23],

$$P = \frac{\text{tp}}{\text{tp} + \text{fp}} \qquad R = \frac{\text{tp}}{\text{tp} + \text{fn}} \qquad F_1 = \frac{2PR}{P + R} \tag{5}$$

In other words, the F_1 expresses the harmonic mean between the correctness (i.e., precision) and the completeness (i.e., recall) of coherent/non-coherent methods classification. In case the two classes of samples are very unbalanced, and they are computed for the larger class, both precision and recall, and thus F_1, are very close to 1, while they have more significant values for the other class. This can be observed in Table VIII for the two versions of JFreeChart. In the optimal case, described in the following, the value of F_1 computed with respect to coherent methods is large, while it is unacceptably low with respect to the non-coherent methods. As for JHotDraw, where non-coherent methods are much more than the coherent ones, the relationship is reversed, and F_1 is greater for non-coherent methods than for the coherent ones. Eventually, the two values of F_1 are comparable for CoffeeMaker, where both classes have a small number of methods.

In an application which constantly monitors comments to detect cases of non-coherence, the number of these should



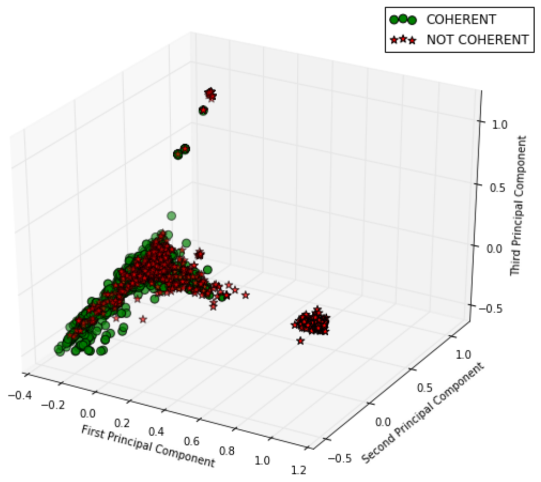


Fig. 7: Visualization of data in the two and three dimensional feature spaces obtained by applying the Principal Component Analysis (PCA) to vectors representing the documents in the VSM.

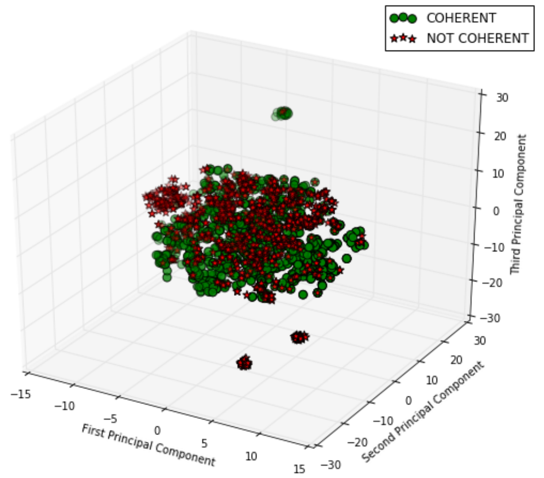


Fig. 8: Data visualization of a three dimensional feature space resulting by the application of the t-SNE.

be much lower than the others, and the F_1 measure for non-coherent methods should give a sufficiently clear assessment of the application. However, in our case, we are considering applications as they are, and situations are quite different from one and another. Therefore, the value of F_1 referred to only one of the classes is not sufficient to assess the performance, and we compute their average.[10]

To evaluate whether an approach simply based on lexical similarity can be effective in separating coherent and non-coherent methods, we evaluated the maximum value of F_1 which can be obtained by choosing different values for the threshold on lexical similarity. More in detail, we considered 100 equally spaced values in the interval $[\mu_n - \sigma_n, \mu_c + \sigma_c]$, that is, in the interval starting a standard deviation before the mean of the similarity values for non-coherent methods and ending a standard deviation after the mean for coherent methods: indeed, it would not be reasonable to choose a threshold out of this interval. We then computed the macro-average F_1 obtained by all these values and took the maximum among them: this represents the best performance which could be obtained with this approach when an oracle gave us the optimal threshold value. Resulting performance is reported in Table VIII.

Even in the optimal case, the value of F_1 is too low for any effective application, meaning that the distinction between coherent and non-coherent pairs of method implementation and comment can not be only based on lexical similarity. In other words, by concentrating all the input description in a unique feature we loose too much information. As an alternative, we now consider approaches of feature reduction to see whether a small number of features can be effective in the identification of coherent or non-coherent methods.

B. *Feature reduction*

The VSM as described above results in $5,642$ features. In order to study the data, we need to reduce them to two or three feature so that we can visualize them. Although the reduction in three dimensions obviously involves lack of information, it can give an idea of how confused the two classes are.

Principal Component Analysis (PCA) [4] is the most widely known method of feature reduction which is usually applied to obtain data visualization. The resulting data are reported in Figure 7. It is not easy to distinguish coherent from non-coherent methods. An alternative approach to feature reduction for visualization can be based on t-Distributed Stochastic Neighbor Embedding (t-SNE) [34]. An analysis of the results reported in Figure 8 gets to the same conclusion.

[10]Such approach is usually referred to as *macroaveraging* [23].

C. Classification

As the last step in our study, we applied a supervised learning classifier leveraging on the evaluations on the coherence of methods included in our dataset. In more details, we adopted the Support Vector Classification (SVM) [35] technique because of its widely recognized generalization properties and its good performance on very different tasks [4], [5]. To avoid issues related to data sparsity, we consider all the methods together, regardless the software they belong to. Methods are then randomly split in test and training sets, respectively corresponding to one and three fourths of the data. Several parameters (also know as *hyper-parameters* [2]) need to be set before we can train the classifier. In our case, these parameters include the type of kernel to be used and its corresponding settings [35]. To this end, we applied a 10-fold cross-validation on the training set in combination with a Grid-Search algorithm [2]. The best setting results to be based on the Radial Basis Function (RBF) Gaussian kernel, with $\gamma = C = 1.0$.

When applied on the test set, we obtain an accuracy of 83.50%, and an average F_1 of 86.99% which are both more than acceptable, and surely better than the one obtained by only using lexical similarities (see Table VIII). In Table IX we report the classification results we obtained on the test data, considering the entire set (i.e., "All") and samples extracted for each application in the dataset. Interestingly, JHotDraw, which showed to be the most difficult to annotate and obtained the worst agreement among human judges, as reported in Table IV, is also the most difficult for the automatic classifier and is characterized by the lowest F_1 score. For the sake of completeness, we also report that the area under the precision-recall curve (AUC) [23] has a value of 81.34%.

D. Discussion

Results derived in our study suggest that the lexical similarity alone is not sufficient to automatically determine whether a lead comment and its corresponding implementations are coherent. In fact, we observed that the lexical similarity is quite low in both coherent and non-coherent cases, even if it resulted to be slightly greater in case of coherent methods. To corroborate this evidence, we applied the Mann Whitney test on the similarities for coherent and non-coherent methods. We verified that the two sets of values correspond to different statistical distributions. Therefore a threshold on the similarity value could be imposed to distinguish coherent methods from non-coherent ones. In this respect, we then applied a macro-average approach to determine the value for this threshold which maximizes the F_1 metric. However, even in the optimal case, results are too low for any effective application, thus confirming that the distinction between coherent and non-coherent methods can not solely rely on lexical similarities.

As an alternative, we then considered approaches of feature reduction but we did not obtain relevant results that could suggest us to further investigate in this direction.

Finally, we exploited the coherence annotations in our dataset to set up a supervised machine learning pipeline based on the application of the SVM classifier. The conclusion we can draw from this preliminary classification test is that the approach is viable. Results can be probably improved by a more accurate feature engineering and a more extensive experimentation on different classification approaches. This represents one of the most important directions for our research.

All the code, the data and obtained results have been collected in a repository publicly available on Github[11] for replication purposes.

TABLE IX: Average $F1$ Classification Results

Application	Accuracy	Average $F1$
CoffeeMaker	90.00%	92.31%
JFreeChart 6.0	81.74%	89.12%
JFreeChart 7.1	85.33%	91.13%
JHotDraw	83.19%	83.73%
All	83.50%	86.99%

VI. Final Remarks

We have presented some results of a research on the coherence between lead comment of methods and their corresponding implementations. Our research started on September 04^{th} 2013 and is still in progress. In particular, we have provided a description of the problem settings, along with the protocol defined to create our dataset. This dataset is publicly available on the web. We also discussed the results of quantitative and qualitative analysis performed on a set of $3,636$ methods. We also investigated the effectiveness of adopting a Vector Space Model with the *tf-idf* schema to discriminate coherent and non-coherent methods. We concluded that the lexical similarity alone is not sufficient for this distinction, while encouraging results have been obtained by applying a Support Vector Machine classifier on the dataset.

There could be many possible future directions for our research. We can sketch some of them as follows:

1) It would be interesting to conduct an empirical study to investigate how maintenance operations affect coherence of methods in multiple versions of the same software. For example, do developers pay attention to out-of-sync methods? In other words, do developers modify a comment conveying information unrelated and/or inconsistent with source code to which it refers to? As a step forward this direction, we already included in the dataset two versions of JFreeChart. Our results and those by Fluri *et al.* [15] represent a viable starting point for future work.
2) We plan to empirically assess how non-coherence between method comment and implementation affects performances of software maintenance and evolution approaches that rely on text search and text retrieval techniques. Possible research questions could be: *(i)* Do these approaches get worse performances when applied

[11] https://github.com/leriomaggio/code-coherence-analysis

to source code containing non-coherent methods? and *(ii)* Do benefits from the use of text search and text retrieval techniques obfuscate from lack of coherence? If the answers to these questions are affirmative, to what extent do non-coherent methods affect search and text retrieval techniques applied to software engineering?

3) We also plan to conduct controlled experiments with human participants to assess the effect of non-coherent methods on source code maintainability and comprehensibility. We plan to involve both Master and Bachelor students as well as professional developers from our contact area network. This part of our research is still in progress and represents one of the most important and challenging one.
4) Since lack of coherence may indicate either a fault in a method body or a fault in its comment [33], we also plan to conduct an empirical study on software repositories to understand if this issue might mislead method callers so introducing faults in their source code. Results from this study might have important practical implications.

Acknowledgment

We would like to thank the annotators of our dataset and the reviewers for their precious and constructive comments and suggestions.

References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Procedings of the International Conference on Software Maintenance*, pages 40–51. IEEE Computer Society, 2000.

[2] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012.

[3] D. Binkley, D. Lawrie, L. Pollock, E. Hill, and K. Vijay-Shanker. A dataset for evaluating identifier splitters. In *Proceedings of International Conference on Mining Software Repositories*. IEEE Computer Society, May 2013.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] I. Campbell and Y. Yiming. *Learning with Support Vector Machines*. Morgan and Claypool, 2011.

[6] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings of International Conference on Software Maintenance*, pages 97–107. IEEE Computer Society, 2000.

[7] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[8] J. Cohen. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological Bulletin*, 1968.

[9] A. Corazza, S. Di Martino, and V. Maggio. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *Proceedings of International Conference on Software Maintenance*, pages 233–242. IEEE Computer Society, 2012.

[10] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. Investigating the Use of Lexical Information for Software System Clustering. In *Proceedings of European Conference on Software Maintenance and Reengineering*, pages 35–44. IEEE Computer Society, 2011.

[11] A. Corazza, V. Maggio, and G. Scanniello. On the coherence between comments and implementations in source code. In *Proceedings of EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 76–83. IEEE Computer Society, 2015.

[12] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the International Conference on Design of communication: documenting & designing for pervasive information*, pages 68–75. ACM, 2005.

[13] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 183–192. ACM, 2005.

[14] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *J. Softw. Maint. Evol.: Res. Pract.*, page n/a, 2011.

[15] B. Fluri, M. Wursch, and H. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the Working Conference on Reverse Engineering*, pages 70–79. IEEE Computer Society, 2007.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] R. J. Freund and W. J. Wilson. *Statistical methods (2. ed.)*. Academic Press, 2003.

[18] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In S. Diehl, H. Gall, and A. E. Hassan, editors, *Proceedings of Mining Software Repositories*, pages 179–180. ACM, 2006.

[19] J. Keyes. *Software Engineering Handbook*. Taylor & Francis, 2002.

[20] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243, 2007.

[21] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501. ACM, 2006.

[22] D. Lawrie, D. Binkley, and C. Morrell. Normalizing Source Code Vocabulary. In *Proceedings of Working Conference on Reverse Engineering*, pages 3–12. IEEE Computer Society, 2010.

[23] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, USA, 2008.

[24] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Trans. Software Eng.*, 38(5):1069–1087, 2012.

[25] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, Dec. 2004.

[26] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[27] F. Salviulo and G. Scanniello. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In *Proceedings of International Conference on Evaluation and Assessment in Software Engineering*, pages 423–432. ACM Press, 2014.

[28] G. Scanniello, A. Marcus, and D. Pascale. Link analysis algorithms for static concept location: an empirical assessment. *Empirical Software Engineering*, 20(6):1666–1720, 2015.

[29] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 21–. IBM Press, 1997.

[30] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, Sept. 1984.

[31] D. Steidl, B. Hummel, and E. Jürgens. Quality analysis of source code comments. In *Proceedings of International Conference on Program Comprehension*, pages 83–92. IEEE Computer Society, 2013.

[32] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the Symposium on Operating Systems Principles*. ACM, October 2007.

[33] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of International Conference on Software Testing*, pages 260–269. IEEE Computer Society, 2012.

[34] L. Van Der Maaten. Accelerating t-sne using tree-based algorithms. *J. Mach. Learn. Res.*, 15(1):3221–3245, Jan. 2014.

[35] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[36] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer, 2012.