Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment

Andrea De Lucia[1], Massimiliano Di Penta[2], Rocco Oliveto[1] and Francesco Zurolo[1]

[1] *Department of Mathematics and Informatics, University of Salerno, Italy*
[2]*RCOST – Research Centre on Software Technology, University of Sannio, Benevento, Italy*
adelucia@unisa.it, dipenta@unisannio.it, roliveto@unisa.it, frazur@gmail.com

Abstract

The presence of traceability links between software artefacts is very important to achieve high comprehensibility and maintainability. This is confirmed by several researches and tools aiming at support traceability link maintenance and recovery. We propose to use traceability information combined with Information Retrieval techniques within an Eclipse plug-in to show the software engineer the similarity between source code components being developed and the high level artefacts they should be traced on. Such a similarity suggests actions aiming at improving the correct usage of identifiers and comments in source code and, as a consequence, the traceability and the comprehensibility level. The approach and tool have been assessed with a controlled experiment performed with master students.

Keywords: traceability recovery, empirical studies.

1. Introduction

Traceability is the mechanism that allows to create links between and within software artefacts. The IEEE Glossary of Software Engineering [22] defines the traceability as:

"*The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match".*

Ensuring traceability across software artefacts is important for different reasons. Traceability helps in software comprehension, allowing to map high-level documents, and thus abstract concepts, to low-level artefacts. This also improves the software maintainability: once a maintainer has identified the high-level document (e.g., requirement, use case) related to the feature to be changed, traceability helps to locate the pieces of design, code and whatever need to be maintained.

The importance of maintaining traceability links is confirmed on one side by the support provided by many CASE tools (see for instance Rational Requisite Pro [29]) and on the other side by the methods and tools presented in the literature to recovery traceability links. Some of these approaches exploit Information Retrieval (IR) techniques [5], [15] to recover traceability links between software artefacts [1], [4], [8], [13], [20], [21], [23], [31], by relying on the hypothesis that there is a correct use of domain terms between artefacts to be traced. In case this does not happen, traceability recovery fails to be effective.

This paper proposes the use of traceability information combined with IR techniques from a different perspective. When a programmer writes the source code, a plug-in incorporated in the development environment shows the similarity between the source code under development and high-level artefacts on which the source code should be traced. Such a similarity provides information about the consistency between source code identifiers and high-level artefacts, suggesting the developer that the code is (or is not) properly traced to the related artefacts.

In the second case, the developer can act in various ways. First, he/she can try to make source code identifiers more consistent. Second, he/she can better comment source code: it is well-known that comments constitute a valuable source of information for many program comprehension tasks. This would help to produce code that (i) is better traceable in case the existing traceability links are lost after maintenance interventions and (ii) contains better identifiers and

Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)
0-7695-2601-2/06 $20.00 © 2006 IEEE

comments, improving comprehensibility and maintainability.

The contribution of this paper can be summarised as follows:

1. the paper describes an Eclipse [17] plug-in, called COCONUT (COde COmprehension Nurturant Using Traceability) that uses Latent Semantic Indexing (LSI) to indicate the similarity between the source code under development and a set of high-level artefacts. The plug-in works with the Java Development Tool (JDT) [17], although it can be easily adapted for other Eclipse tools (e.g., UML modellers);
2. the paper reports results of a controlled experiment performed to assess the usefulness of the proposed approach and tool. The experiment involves master students in development tasks with and without the use of the traceability plug-in, and uses the similarity measure between code and high-level artefacts as a measure of the traceability level achieved;
3. the paper reports results of a code inspection made after the experiment with the purpose of assessing the quality of code identifiers and comments.

The paper is structured as follows. Section 2 describes the COCONUT plug-in and the underlying information retrieval method. Section 3 describes the controlled experiment and discusses the obtained results. After a discussion of the related literature (Section 4), Section 5 concludes and outlines directions for future work.

2. Traceability based similarity improvement tool

In this Section we describe how an IR approach, based on LSI [15], can be used to compute the similarity between software artefacts and present an Eclipse plug-in that relies on this approach to highlight the similarity between the source code under development and high-level artefacts to which the source code should be traced.

2.1. Artefact similarity computation

An IR method compares a given query against all the documents in a collection and ranks these documents according to their similarity with the query. In our case a query is represented by a source code artefact, while a document collection is represented by an artefact repository. Artefacts are indexed in an artefacts space by extracting information about the occurrences of terms within them. This information is used to define similarity measures between pairs of artefacts. The extraction of the terms is preceded by a text normalisation phase performed in three steps:

1. white spaces and most non-textual tokens from the text (i.e., operators, special symbols, some numbers, etc.) are pruned out;
2. source code identifiers composed of two or more words are split into separate words (i.e., *TelephoneNumber* and *telephone_number* are split into the words *telephone* and *number*);
3. all the terms in the stop word list or having a length less than three are removed.

To determine the similarity between query and documents, we use LSI, an extension of the Vector Space Models (VSM), that was developed to overcome the synonymy and polysemy problem [15]. LSI explicitly takes into account the dependencies between terms and between artefacts, in addition to the associations between terms and artefacts. LSI assumes that there is some underlying or "latent structure" in word usage that is partially obscured by variability in word choice, and use statistical techniques to estimate this latent structure. For example, both "car" and "automobile" are likely to co-occur in different artefacts with related terms, such as "motor", "wheel", etc. Finally, LSI exploits information about co-occurrence of terms (latent structure) to automatically discover synonymy between different terms.

LSI defines a term-by-document $m \times n$ matrix where m is the number of terms, and n is the number of indexed artefacts. A generic entry $a_{i,j}$ of this matrix denotes a measure of the weight of the i^{th} term in the j^{th} artefact. Then it applies Singular Value Decomposition (SVD) [11], a technique used to project the original term-by-document matrix into a reduced space in order to diminish the obscuring "noise" in word usage. Thus, the retrieval is based on the semantic content of the documents rather than their lexical content. As a consequence, a relevant document may be retrieved even if it does not share any literal terms with the query. The cosine of the angle between the reduced artefact vectors gives the similarity between the corresponding artefacts.

2.2. Plug-in description

We have implemented a tool, called COCONUT, that shows the similarity between the source code

Figure 1 – COCONUT view

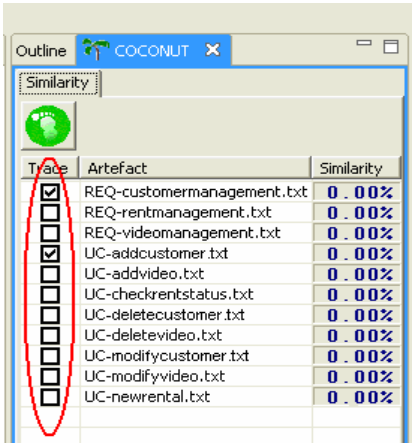


Figure 2 – Selection of the high level artefacts traced onto the source code under development

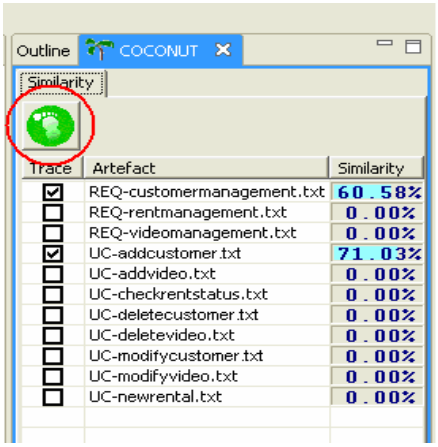


Figure 3 – Visualisation of the similarity level between the source code under development and the related high level artefacts

under development and related high-level artefacts as a plug-in for the Eclipse Platform.

The plug-in contributes a new view to the Eclipse workbench, displaying a sorted list of all indexed artefacts as a table (see Figure 1). The first column of the table contains a check box that indicates if the artefact is traced onto the source code under development. The second column contains the description of the high level artefacts, while the third column represents the similarity value between the artefact and the source code under development. It is worth noting that the list of indexed artefacts is extracted from the artefact space. Using the plug-in preferences the user can choose if he/she wants to create a new artefact space, or import an artefact space in his/her project workspace. In the latter case he/she selects the artefact space to import represented by a term-by-document matrix stored in a XML file.

Figure 1 and Figure 2 show a scenario where the developer is coding the Java class *AddCustomer* (see Figure 1). During development he/she can use COCONUT to visualize the similarity between the class under development and the related high-level artefacts. To this aim he/she selects the artefacts related to *AddCustomer*, namely the requirement *REQ-customermanagement.txt* and the use case *UC-addcustomer.txt* (see Figure 2) and clicks on the button in the top of the plug-in view (see Figure 3). Analysing the similarity, he/she can have an indication of how the source code identifiers and comments are consistent with the related high-level artefacts.

3. A controlled experiment to assess the usefulness of the traceability plug-in

This section describes a controlled experiment that has been carried out to empirically assess the plug-in described in this paper. The experiment is described following the template proposed by Wohlin *et al.* [34].

3.1. Experiment definition and context

The objective of the experiment is to analyse the similarity between high-level artefacts and source code (with and without comments) with the purpose of evaluating the usefulness of an Eclipse plug-in, COCONUT, that indicates how requirements are well-traced to the source code identifiers and comments. The quality focus is to ensure high traceability as well as consistency in the identifiers and well-commented code, while the perspective is both (i) of a researcher, that wants to evaluate how suggestions based on traceability information help the developer to use meaningful identifiers; and (ii) of a project manager, that wants to evaluate the possibility of adopting the plug-in within his/her own organisation.

The context of the experiment is constituted of 16 computer science 2nd year master degree students (having a comparable level of ability), grouped in 8 pairs, each one constituting a development team. In the context of this experiment, subjects will have to perform two evolution tasks over the ADAMS software system [12]. ADAMS is an artefact-based process support system developed by graduate students at the University of Salerno. It has been implemented as a web-based system using Java technologies, Apache Tomcat as web server and MySQL as Database Management System. The user interface of the system is implemented by 90 Java Server Pages. The code for the application logic and data layer is composed of about 40K lines of java code, spread among 40 servlets implementing the application logic subsystems and 92 java beans providing data layer functionalities. The database is composed of 22 database tables. Although ADAMS is a system developed by graduate students, it is large and complex enough to be considered a real system rather than a toy program.

The experiment is performed online, i.e., in a controlled laboratory setting. The two evolution tasks students have to perform deal with:

1. creating a part of a web application for listing, inserting and modifying comments related to artefacts managed through ADAMS;
2. creating a module able to handle templates of documents (e.g., forms) to be managed through ADAMS.

3.2. Hypotheses formulation

As said before, the experiment aims to assess whether the use of the COCONUT plug-in during development or maintenance task helps to improve the use of correct identifiers and comments, and, because of that, ensures high traceability by improving the similarity between source code and high-level artefacts. Moreover, we are interested to see how this similarity is influenced by comments inserted by developers into source code. Consequently, the two null hypotheses can be formulated as follows:

- **H_{01}:** the use of the COCONUT plugin does not significantly affect the similarity between the source code (including comments) maintained /developed and the related requirements;
- **H_{02}:** the use of the COCONUT plugin does not significantly affect the similarity between the source code (excluding comments) maintained /developed and the related requirements;

while the alternative hypotheses are:

- **H_{a1}:** the use of the COCONUT plugin significantly affects the similarity between the source code (including comments) maintained /developed and the related requirements;
- **H_{a2}:** the use the COCONUT plugin significantly affects the similarity between the source code (excluding comments) maintained /developed and the related requirements.

3.3. Variable selection and experiment design

In order to properly design the experiment and analyze the results, the following independent variables need to be considered:

- ***Method*:** this variable indicates the factor on which the study is focused, i.e., performing a maintenance task using the COCONUT plug-in (*PL*) or performing the task without using the COCONUT plug-in (*NOPL*).
- ***Task*:** i.e., the evolution task performed over ADAMS. As described above, the experiment involves two tasks, named *T1* and *T2*.
- ***Lab*:** the experiment is organised in two laboratory sessions, indicated with *Lab1* and *Lab2*.
- ***Comments*:** we need to understand if the plug-in helps developers to improve code traceability with (*C*) and without (*NC*) considering comments introduced in the source code.

We did not consider students' ability as a factor for blocking, since we assumed ability to be comparable across students and across teams. The dependent variable is the *Similarity* between the maintained code and the related requirement(s), measured as the cosine of the angle between source code and requirement vectors (see Section 2.1).

Table 1 – Experiment Design

	Group			
	A	**B**	**C**	**D**
Lab1	T1_PLL	T1_NOPL	T2_PLL	T2_NOPL
Lab2	T2_NOPL	T2_PLL	T1_NOPL	T1_PLL

Given the mentioned variable, the experiment design is reported in

Table 1. Possible treatments are all combinations of factors *Method* (*PL* and *NOPL*) and *Task* (*T1* and *T2*). To avoid results to be biased by group ability, each group will experience both *Methods* and both *Tasks* over two subsequent *Labs*. Also, to minimize the learning effect, we need to have groups starting to work in *Lab 1* with or without plug-in on both tasks. All these considerations make necessary to group teams into four groups: *A* (teams 1 and 2), *B* (teams 3 and 4), *C* (teams 5 and 6), *D* (teams 7 and 8).

3.4. Preparation

Students were properly selected to allow creating uniform groups as explained in the previous section and avoid the need for blocking.

Training laboratories were carried out to give students an equal, prior knowledge of the system to be maintained (i.e., ADAMS). Also a tutorial laboratory was performed to introduce the traceability plug-in and let subjects get confidence with it through some examples of code and requirements (not related to ADAMS to avoid biasing the experiment). Finally, right before the experiment, we showed to students a presentation with detailed instructions on the tasks to be performed.

3.5. Experiment material and execution

Each student was provided with the following material:

1. handouts of the introductory presentation and the plug-in user manual;
2. requirement documents and use case descriptions for the task to be performed;
3. the ADAMS source code;
4. the Eclipse/JDT environment (to perform the maintenance task) with or without the COCONUT plug-in installed (depending on the treatment); the Eclipse/JDT environment (to perform the maintenance task) with or without the COCONUT plug-in installed (depending on the treatment);
5. a survey questionnaire to be filled after each *Lab*. The questionnaire (shown in Table 2) was composed of questions expecting closed answers according to the Likert scale [26] – from 1 (*strongly agree*) to 5 (*strongly disagree*) – to assess how the lab and task were clear, if subjects had enough time, and other related questions. In addition, for subjects using the plug-in, the survey investigates on the usefulness of the plug-in and ease of use, and asks how much time subjects spent using it.

For each *Lab*, subjects had 2 hours available to perform the required *Task*. After the task was completed, they sent us by email an archive containing the classes maintained and returned the filled survey questionnaire.

3.6. Experiment results

After the experiment execution, artefacts produced/maintained by teams were collected, and similarity between source code and high-level artefacts was computed, considering code with and without comments. Table 3 reports descriptive statistics of the obtained similarity levels, grouped by *Task*, *Method* and use of *Comments*. Results are also summarised as box plots in Figure 4.

Since each subject performed a maintenance task with and without the use of the plug-in, the null hypothesis has been tested using a paired, non-parametric test[1], i.e., the Wilcoxon paired test [10]. Table 4 reports the results (p-values and descriptive statistics over the differences) of the test, with a significance level of 95%. As it can be seen, while considering comments, the null hypothesis (H_{01}) can be rejected, nothing can be said when analysing the code without comments, i.e. the null hypothesis H_{02} cannot be rejected.

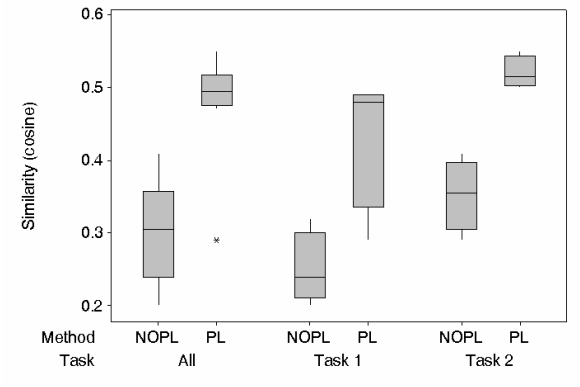
Table 2 – Post-Experiment Questionnaire

Id	Question
1	I had enough time to perform the lab task
2	The objectives of the lab were perfectly clear to me
3	The task I had to perform was perfectly clear to me
4	The requirement given to me provided enough information to perform the required task
5	I was able to locate the classes I had to maintain
6	The use of the traceability tool was clear to me
7	I found the traceability tool useful
8	How much time (in terms of percentage) did you spend using the traceability tool during this lab (A < 25% - B >= 25% and < 50% - C >= 50% and < 75% - D >= 75%)?

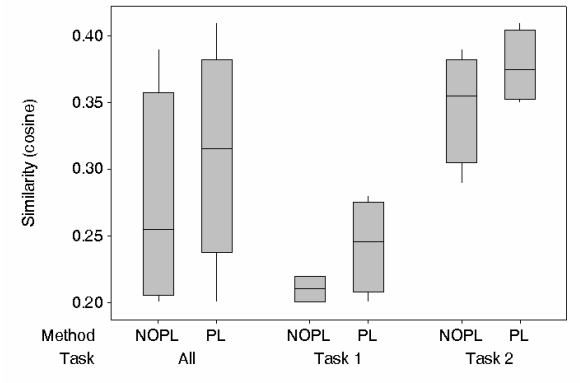
[1] Due to the number of samples available, parametric statistics cannot be used.

Table 3 – Descriptive statistics

Task and Method		With Comments			Without Comments		
		Median	Mean	Std. Dev.	Median	Mean	Std. Dev
All	PL	0.49	0.48	0.079	0.31	0.31	0.078
	NOPL	0.30	0.30	0.071	0.25	0.28	0.079
1	PL	0.48	0.43	0.097	0.24	0.24	0.035
	NOPL	0.24	0.25	0.050	0.21	0.21	0.011
2	PL	0.51	0.52	0.021	0.37	0.28	0.027
	NOPL	0.35	0.35	0.049	0.35	0.35	0.042



(a)



(b)

Figure 4 – Similarity by method and task with comments (a) and without comments (b)

A two-way Analysis of Variance (ANOVA) [16] was used to investigate on the effect of the different factors. Results are summarised in Table 5. They indicate that, when considering comments, there is a dependency on both *Method* and *Task*, while the interaction between the two factors is not significant.
When removing comments (Table 6), *Task* has even more influence than method, that only have a slighter effect (p-value=0.068). Further ANOVA analyses revealed no significant dependency of the similarity on the *Lab* (indicating absence of learning effect) or on the particular team considered. Other, non-parametric tests such as the Friedman test confirmed the ANOVA results.

The analysis of data collected from survey questionnaires (see Figure 5) showed that the objectives, the laboratory tasks and the requirements provided were clear, and that it was easy to locate classes to be maintained. However, results showed (with p-value 0.011 computed with Mann Whitney U-test) that subjects performing *Task T1* felt to have significant less time than subjects performing *Task T2*. Questions related to the use of the plug-in (that occupied between 25% and 50% of the laboratory time) revealed that the use of the plug-in was clear. However, subjects involved in *Task T2* using plug-in perceived a higher usefulness (p-value 0.02). This contradicts the other analyses performed by *Task*: such analyses revealed that *Task T2* exhibits a smaller difference between *PL* and *NOPL*. The explanation can be found in the fact that, as shown in Figure 4, the similarity level for *Task T1* (due to the characteristics of the class code) is intrinsically lower than for *Task T2*. This made students unable to achieve high similarity levels, and caused them a perception that the plug-in was a bit less useful. Nevertheless, in the specific case of a task longer to be performed (hence a bit more complex) such as *Task T1*, the plug-in helped more than in simpler cases (e.g., *Task T2*) to gain traceability.

Table 4 – Wilcoxon paired test p-values and descriptive statistics of differences (by team)

Comment	Median	Mean	Std. Dev.	p-value
Yes	0.16	0.18	0.12	0.011
No	0.035	0.031	0.15	0.20

Table 5 – ANOVA table of similarity by Method and Task (using comments) R^2= 78.18%, R^2(adj) = 72.73%

Source	DF	Sum of Squares	Mean Square	F	p-value
Method	1	0.124	0.124	33.45	<10e-8
Task	1	0.0352	0.352	9.46	0.010
Interaction	1	0.00031	0.00031	0.08	0.779
Residual	12	0.0446	0.00371	-	-
Total	15	0.204	-	-	-

Table 6 – ANOVA table of similarity by Method and Task (without comments) R^2= 87.05%, R^2 (adj) = 83.82%

Source	DF	Sum of Squares	Mean Square	F	p-value
Method	1	0.00391	39.063	4.03	0.068
Task	1	0.0743	0.0743	76.65	<10e-8
Interaction	1	0.000006	0.000006	0.01	0.937
Residual	12	0.0116	0.000969	-	-
Total	15	0.0898	-	-	-

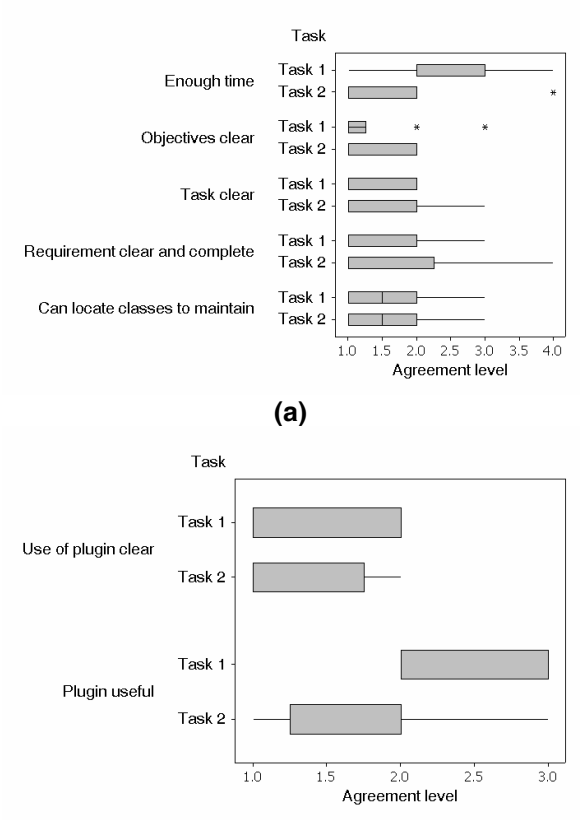


Figure 5 – Survey Questionnaire: generic questions (a) and questions related to the use of the plug-in (b)

3.7. Inspecting the produced code to assess the use of identifiers and comments

The results of the controlled experiment indicate that the use of the tool helps to improve the similarity between code and related requirements, especially when considering code comments. This, however, does not provide precise insights about the quality improvement of the produced code, and in particular, on the proper use of identifiers and comments.

To this aim, three authors inspected the source code produced by subjects (without being aware of the *Method* used for each code artefact), answering the inspection checklist shown in Table 7 (possible answers are 1: *Accept*; 2: *Minor revision*, 3: *Major revision*, 4: *Reject*). The checklist is focused on the proper use of naming conventions, on the use of informative identifiers and on the correct use of comments. After code inspection, an inspection meeting was used to resolve evaluation conflicts. Results are summarised as box plots in Figure 6.

Answers to questions related to comments indicate that the use of the COCONUT plug-in introduced an overall improvement to comments (C1), to method signature comments (C3), and to statement comments (C4). Overall, students tended not to comment variable declarations (C4), while a large variability was seen for class level comments (C2). However, in the last case the plug-in does not really help, since anyone could comment the class purpose starting from the task description.

With regards to the naming of identifiers, the only case in which the plug-in helped was for improving the naming of local variables (I3). The last case is the one where, for our case studies, the use of the plugin is really relevant to help improving identifiers, since classes produced in the two tasks were servlets, having a few (or no) instance variables and constants, and also methods besides the standard servlet methods (*doGet*, *doPost*, etc.) were, in general, limited.

Table 7 – Code inspection checklist

Id	Question
I1	Methods are properly named
I2	Instance variables are properly named
I3	Local variables are properly named
I4	Constants are properly named
C1	Overall, code is properly commented
C2	Class-level comments
C3	Method purpose and signatures are explained
C4	Instance variables are properly commented
C5	Logically-related statements are commented together
C6	Declarations of variables with unclear names are properly commented

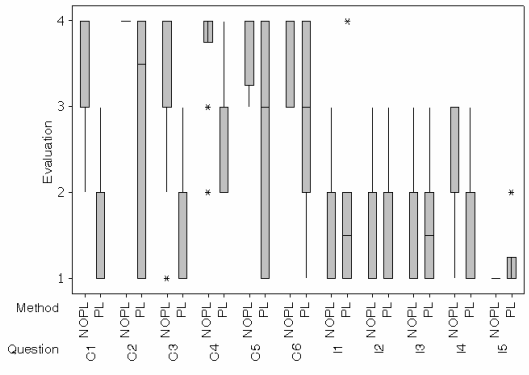


Figure 6 – Code inspection results

3.8. Threats to validity

This Section describes the threats to validity that can affect our experiment: internal, construct, conclusion and external validity threats.

Regarding *internal validity* threats, the experiment design mitigates the threats due to maturation or learning effects, since each group worked, over the two *Labs*, on different *Tasks* and with two different *Methods* (*PL* and *NOPL*). While subjects may have learnt across *Labs* how to make identifiers consistent, ANOVA showed no such a significant effect. There was no abandonment, and as shown in the survey everything was clear. Despite the fact that students used the similarity to improve traceability, they did not know exactly the hypothesis of the experiment. Finally, students were not evaluated on their performance in the *Lab*.

Construct validity threats that may be present in this experiment, i.e., interactions between different treatments, were mitigated by a proper design that allowed to separate the analysis of the different factors and of their interactions. The measurement instrument, i.e. the similarity, is the same used in traceability recovery, that would benefit how having consistent identifiers and comments into source code. Survey questionnaires, mainly indented to get qualitative insights, were designed using standard ways and scales [26]. Regarding the code inspection, clearly the evaluation can have some degree of subjectivity. To mitigate this, three independent evaluations were made and an inspection meeting was used to discuss and resolve high discrepancies. Bias was avoided by accessing to files without having knowledge on the *Method*. It is finally important to note that the code inspection was mainly thought to give us some qualitative insights rather than empirical evidence.

About *conclusion validity*, proper tests were performed to statistically reject null hypotheses. In cases where differences were present but not significant, this was explicitly mentioned. Non-parametric tests were used in place of parametric tests where there were not the conditions necessary to use parametric tests. Also, ANOVA results were confirmed by non-parametric tests (Friedman test).

Finally, *external validity* threats are always present when experimenting with students. Last-year master students have a very good analysis, development and programming experience, and they are not far from junior industry programmers. Also, the system to maintain and the task to be performed were reasonably complex. Nevertheless, it is worth replicating the experiment with industry developers and other systems, to confirm or contradict the obtained results.

4. Related work

Since the importance of the artefact traceability management has been acknowledged, several research and commercial tools to support it have been proposed. Some of them only provide an effective support for recording, displaying, and checking the completeness of installed traces. Examples are: DOORS [32], Rational RequisitePro [29], RDD.100 [19], gIBIS [9], TOOR [27], and REMAP [28]. Recently, the artefact traceability support has been introduced in some process support systems where the traceability layer has been combined with event-based notifications mainly to make users aware of artefact modifications [7], [12]. All these tools have a common drawback: they require a manual maintenance of the traceability layer while the system changes and evolves. This is the reason why in the papers [13], [14] the authors proposed to integrate a traceability recovery tool (based on a IR technique) in the ADAMS system [12] to support the software engineer during the traceability link identification and maintenance.

The traceability recovery problem is widely tackled in the literature and several techniques are applied to support the process of traceability link recovery. Some of them deal with recovering traceability links between design and implementation. The proposed approaches represent source code and high-level models using a common language and use regular expressions [25], maximum match algorithm [3] or more tolerant string matching [33] to map source code in the high-level models.

Other approaches consider text documents written in natural language, such as requirements documents. Zisman *et al.* [35] automate the generation of traceability relations between textual requirement artefacts and object models using heuristic rules. Such rules are used to match syntactically related terms in the textual parts of the requirements artefacts with related elements in an object model and create traceability relations when a match is found.

Recently, several authors have applied IR methods [5], [18], [15] to the problem of recovering traceability links between software artefacts. In particular, Antoniol *et al.* [1] use IR methods based on probabilistic models and VSM [5], [18]. They apply and compare the two methods on two case studies to trace C++ source code onto manual pages and Java code to

functional requirements, respectively: the results show that the two methods have similar performances when a preliminary morphological analysis (stemming) of the terms contained in the software artefacts is performed. In [2] the vector space model is used to trace maintenance requests on software documents impacted by them. Antoniol *et al.* [4] also discuss how a traceability recovery tool based on the probabilistic model can improve the retrieval performances by learning from user feedbacks, provided as input to the tool in terms of a subset of correct traceability links (training set).

Several authors use LSI [15] to recover traceability links between source code and documentation [23] and between different type of artefacts (use cases, interaction diagrams, test cases and code classes) [13]. Marcus and Maletic [23] use LSI to perform the same case studies as in [1] and compare the performances of LSI with respect to VSM and probabilistic models. They demonstrate that, without the need for stemming, LSI can achieve comparable results with probabilistic and vector space models, where stemming are applied.

Huffman Hayes *et al.* [20] use the vector space model to recover traceability links between requirements and compare the results achieved by applying different variants of the base model. In [21] they also address the issues related to improving the overall quality of the requirements tracing process. In particular, they define requirements for a tracing tool based on analyst responsibilities in the tracing process and present a prototype tool, called RETRO (REquirements TRacing On-target), to address these requirements.

Settimi *et al.* [31] investigate the effectiveness of IR methods for tracing requirements to UML artefacts, code, and test cases. In particular, they compare the results achieved applying different variants of the vector space model. Cleland-Huang *et al.* [8] propose an improvement of the dynamic requirements traceability performance introducing different strategies for incorporating supporting information into a probabilistic retrieval algorithm.

In a recent paper Marcus *et al.* [24] discuss in which cases visualising traceability links is opportune, as well as what information concerning these links should be visualised and how. They also present a prototype Eclipse plug-in, called TraceViz, based on the traceability recovery tool presented in [23] to support the software engineer during recovery, visualisation, and maintenance of traceability links. This paper is, in our knowledge, the closest to our work. However, TraceViz does not show the similarity between traced artefacts, but it only classifies traced artefacts in different sets according to the classification proposed in [13]. Also, with respect to the paper [24] our work reports a controlled experiment and results from code inspection, aiming to demonstrate that using traceability information (combined with traceability recovery techniques) during software development can improve the quality and comprehensibility of source code.

5. Conclusion and work-in-progress

This paper proposed a tool, called COCONUT, that supports the user in improving the quality of code identifiers and comments by highlighting the similarity, (computed using Latent Semantic Indexing) between the code under development and related artefacts.

A controlled experiment was performed to assess the usefulness of the plug-in. Results indicate that the plug-in significantly help to improve the similarity between code and related requirements in presence of comments, while a practical (while not always statistically significant) improvement was also seen without considering comments. Code inspection over the artefacts produced during the experiment indicated that the plug-in helps to improve the quality of comments and of variable names.

There are a number of directions to improve the performances of the COCONUT. A first direction is to combine the plug-in with the ADAMS system [12], as the use of the plug-in during software development would be an incentive to inserting and maintaining traceability links as well as increasing the consistency in the use of domain terms within the traced software artefacts. A second direction would be to add a new functionality to the plug-in aiming at suggesting source code identifiers according to the domain terms contained in the corresponding high-level artefacts.

References

[1]. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. "Recovering traceability links between code and documentation", *IEEE Transactions on Software Engineering*, vol. 28, no. 10, 2002, pp. 970-983.

[2]. G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Identifying the Starting Impact Set of a Maintenance Request", *Proc. of 4th European Conference on Software Maintenance and Reengineering*, Zurich, Switzerland, IEEE CS Press, 2000, pp. 227-230.

[3]. G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-Code Traceability for Object Oriented Sys-

tems", *Annals of Software Engineering*, vol. 9, 2000, pp. 35–58.
[4]. G. Antoniol, G. Casazza, and A. Cimitile, "Traceability Recovery by Modelling Programmer Behavior", *Proceedings of 7th Working Conference on Reverse Engineering*, Brisbane, Queensland, Australia, IEEE CS Press, 2000, pp 240-247.
[5]. R. Baeza-Yates, and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison-Wesley, 1999.
[6]. J.Y.J. Chen and S.C. Chou, *"Consistency Management in a Process Environment"*, The Journal of Systems and Software, vol. 47, 1999, pp. 105-110.
[7]. J. Cleland-Huang, C. K. Chang, and M. Christensen, *"Event-Based Traceability for Managing Evolutionary Change"*, IEEE Transactions on Software Engineering, vol. 29, no. 9, 2003, pp. 796-810.
[8]. J. Cleland-Huang, R. Settimi, C. Duan, X. Zou, "Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability", *Proceedings of 13th International Requirements Engineering Conference*, Paris, France, IEEE CS Press, 2005, pp. 135-144.
[9]. J. Conklin and M. L. Begeman, "Gibis: A Hypertext Tool for Exploratory Policy Discussion," *ACM Transactions Office Information Systems*, vol.6, no. 4, 1988, pp. 303–331.
[10]. W.J. Conover, *Practical Nonparametric Statistics,* 3rd Edition, Wiley, 1998.
[11]. J. K. Cullum and R. A. Willoughby, "Lanczos Algorithms for Large Symmetric Eigenvalue Computations", in *Real rectangular matrices*, vol. 1, Brikhauser, Boston, 1985.
[12]. A. De Lucia, F. Fasano, R. Francese, and G. Tortora, "ADAMS: an Artefact-based Process Support System", *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering*, Banff, Alberta, Canada, 2004, pp. 31-36.
[13]. A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Enhancing an Artefact Management System with Traceability Recovery Features", *Proceedings of 20th IEEE International Conference on Software Maintenance*, Chicago IL, USA, 2004, pp. 306-315.
[14]. A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "ADAMS Re-Trace: a Traceability Recovery Tool", *Proceedings of 9th IEEE European Conference on Software Maintenance and Reengineering*, Manchester, UK, 2005, pp. 32-41.
[15]. S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, no. 41, 1990, pp. 391-407.
[16]. J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*, Duxbury, 1999.
[17]. Eclipse Platform, http://www.eclipse.org/
[18]. D. Harman, "Ranking Algorithms", in *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 363–392.
[19]. Holagent Corporation product RDD-100, http://www.holagent.com/new/products/modules.html.
[20]. J. Huffman Hayes, A. Dekhtyar, and J. Osborne, "Improving Requirements Tracing via Information Retrieval", *Proc. of the 11th IEEE Intern. Requirements Engineering Conference*, Monterey, CA, USA, IEEE CS Press, 2003, pp. 138-147.
[21]. J. Huffman Hayes, A. Dekhtyar, S. Karthikeyan Sundaram, and S. Howard, "Helping Analysts Trace Requirements: An Objective Look", *Proceedings of 12th IEEE International Requirements Engineering Conference*, Kyoto, Japan, IEEE Computer Society Press, 2004, pp. 249-261.
[22]. IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society Press, 1991.
[23]. A. Marcus and J.I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing", *Proceedings of the International Conference on Software Engineering*, Portland, Oregon, USA, 2003, pp. 125-135.
[24]. A. Marcus, X. Xie, D. Poshyvanyk, "When and How to Visualize Traceability Links?", *Proceedings of the 3rd ACM International Workshop on Traceability in Emerging Forms of Software Engineering*, Long Beach California, USA, 2005, pp. 56-61.
[25]. G.C. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation", *IEEE Transactions on Software Engineering*, vol.27, no.4, 2001, pp. 364-380.
[26]. A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*, Pinter Publishers, 1992.
[27]. F.A.C. Pinhero and J.A. Goguen, "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, vol. 13, no. 2, 1996, pp. 52–64.
[28]. B. Ramesh and V. Dhar, "Supporting Systems Development Using Knowledge Captured During Requirements Engineering", *IEEE Transactions on Software Engineering*, vol. 9, no. 2, 1992, pp. 498–510.
[29]. Rational RequisitePro web site, http://www.rational.com/products/reqpro/index.jsp.
[30]. G. Salton and C. Buckley, "Term-Weighting Approaches in Automatic Text Retrieval", in *Information Processing and Management*, vol. 24, no. 5, 1988, pp. 513–523.
[31]. R. Settimi, J. Cleland-Huang, O. Ben Khadra, J. Mody, W. Lukasik, and C. DePalma, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts", *Proceedings of 7th International Workshop on Principles of Software Evolution*, Kyoto, Japan, IEEE CS Press, 2004, pp. 49-54.
[32]. Telelogic product DOORS, http://www.telelogic.com.
[33]. J. Weidl and H. Gall, "Binding Object Models to Source Code", *proceedings of the 22nd IEEE Annual International Computer Software and Applications Conference*, Vienna, Austria, IEEE CS Press, 1998, pp. 26–31.
[34]. C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell and A. Wesslen, *Experimentation in Software Engineering – An Introduction*, Kluwer, 2000.
[35]. A. Zisman, G. Spanoudakis, E. Perez-Miñana, and P. Krause, "Tracing Software Requirements Artifacts", *Proc. of International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, USA, 2003, pp. 448-455.