Volume 2, Issue 2, February 2012 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Improving Comprehensibility of Source code By Applying Coding Styles

K. Delhi Babu*
Department of CSE
Sree Vidyanikethan Engineering college

V. Raja sekhar Reddy
Department of CSE
Sree Vidyanikethan Engineering college

Abstract— **Recent studies indicated that computes and shows the similarity between the source code being developed and related high-level artefacts, such as requirements, and usecases, helps developers to improve the quality of source code identifiers. It has been implemented as an Eclipse plug-in, called COde COmprehension Nurturant Using Traceability (COCONUT).In this paper, We presents an approach that applies coding styles to source code for better comprehensibility of source code.**

Keywords— **Software traceability, source code comprehensibility, source code identifier quality, information retrieval, software development environments.**

I. Introduction

Recent studies underline the software quality problems by many approaches and methods, in improving the software quality for supporting developers. Source code textual properties, in particular the usage of proper identifiers, are also an important indicator of software quality. A new cohesion metric (conceptual cohesion), proposed by Marcus et al. , that exploits Latent Semantic Indexing (LSI) to compute the overlap of semantic information in a class expressed in terms of textual similarity among methods.

Consistent use of identifiers and detailed, meaningful comments are two factors that can affect source code maintainability and comprehensibility.

For example, using a non meaningful term referring to a concept or using a different term to refer the same concept may increase the burden of program comprehensibility. And this also leads to error prone because there is a mismatch between the mental model of the developer and intended meaning of the term.

Similarity is an indicator of the quality of source code comments and identifiers between high-level artefacts and the source code. Similarity can be measured by using IR techniques in traceability recovery between text contained in the source code and the domain term contained in high-level software artifacts and suggests that these techniques can also be used to improve identifiers and comments during software development and increase such similarity.

The IR based approach is based on the conjecture—discussed in previous literature [1], [2] and also assumed by traceability recovery approaches—that the similarity is an indicator of the quality of source code comments and identifiers between high-level artifacts and the source code.

In this paper, we demonstrate applying coding styles through SmartFormatter, which implements the proposed approach that learns the coding styles from existing source code and apply these rules to the code under development.

This is paper also describes about COde COmprehension Nurturant Using Traceability (COCONUT), which is an Eclipse plug-in. The COCONUT plug-in computes and shows the similarity between the source code being developed and related high-level artifacts.

This paper extends a previous paper [3]. The paper is structured as follows: After a discussion of IR-Based Traceability Recovery Approach in Section II, Section III provides information about Improving Source Code Lexicon Through COCONUT, Section IV present the proposed approach, Applying Coding Styles, and section V concludes this paper.

II. IR based Traceability Recovery Approach

Information Retrieval is the area of study concerned with searching for documents, for information with in documents, and for metadata about documents, as well as that of searching structured storage, relational databases, and the World Wide Web. IR is the interdisciplinary, based on computer science, library system, information science, physics and statistics.

Traceability is the mechanism that allows to create links between and with in software artefacts. Traceability links between software artifacts have to be identified and maintained during software development and maintenance. It is time consuming process for software developers. So, they need methods and tools for handling traceability links. IR-based methods recover traceability links on the basis of the similarity between the text contained in the software artifacts. The key idea behind such methods is that most of the software documentation is text based or contains textual descriptions, and that programmers use meaningful domain terms to define source code identifiers.

IR methods, includes probabilistic model, Vector Space Models and Latent Semantic Indexing, have also been used to recover traceability between requirements, requirements and design artifacts, and requirements and design documents. In particular, if the source code does not have high similarity with the related high-level artifacts, the quality of source code identifiers or comments is likely to be poor, and this can potentially affect source code understandability and maintainability.

III. Improving Source Code Lexicon Through COCONUT

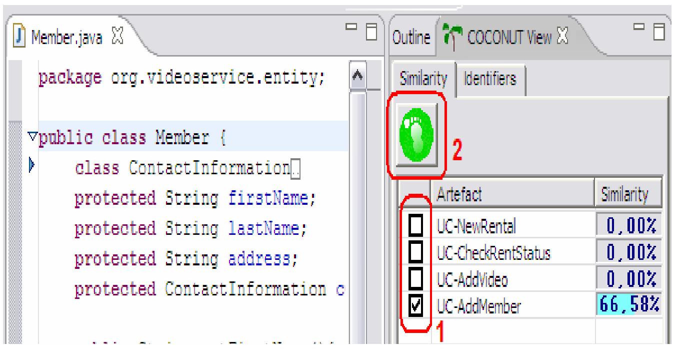
This section describes an approach that improves the quality of source code lexicon under development of software. This approach is based on the conjecture that developers are induced to make the identifiers of source code more consistent with the terms in the high level artifacts or to better comment the source code if the software development environment provides information about the textual similarity between the source code being written and the related high-level artifacts. The information about high level artifacts, for e.g., requirements, module specifications, and use cases, is available during software development.

This approach has been implemented in COCONUT, which is a plug-in for the Eclipse Integrated Development Environment. The COCONUT plug-in works with the Java Development Tool. The Plug-in organized in two different tabs, namely, Similarity and Identifiers. The Similarity tab provides information about the similarity between the source code being written related high level artifacts, while the Identifiers tab suggests appropriate identifiers to be used in the source code under development. The COCONUT plug-in have the following functionalities.

1) *Visualization of the similarity level*: Information about the similarity level between the source code under development and related high level artifacts shows as table by the similarity tab. The first column of the table contains a check box that indicates whether the artifact has to be selected or not and traced onto the source code under development. The second column contains the description of the high-level artifacts, and the third column shows the similarity between the high level artifact and the source code being written. The third column represents the similarity level for the selected high level artifacts.

Similarity represents textual similarity between the source code under development and related high level artifacts. Similarity will be low if there is meaningless identifiers and non related identifiers. Similarity will be high if there is meaningful identifiers and related identifiers. Low similarity represents the comprehensibility of source code is complex where as high similarity represents the comprehensibility of source code is good. Source code comprehensibility influences the source code quality.

The following figure shows the similarity level in the Eclipse IDE using COCONUT.



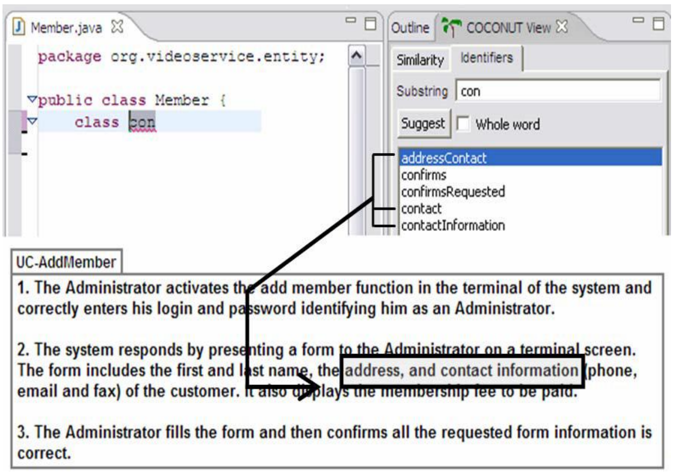
```
package org.videoservice.entity;

public class Member {
    class ContactInformation
    protected String firstName;
    protected String lastName;
    protected String address;
    protected ContactInformation c
```

	Artefact	Similarity
☐	UC-NewRental	0,00%
☐	UC-CheckRentStatus	0,00%
☐	UC-AddVideo	0,00%
☑	UC-AddMember	66,58%

Fig1. The COCONUT view in the Eclipse IDE: Visualization of similarity level

2) *Suggestion of source code identifiers*: The tab Identifiers of the COCONUT view shows a sorted list of candidate identifiers extracted from the related high-level artefacts (selecting in the tab Similarity) and containing a given substring (see Figure 2). To give suggestion by COCONUT the programmer has two possibilities: first, the programmer selects the menu item *Get suggestions* of the pop-up menu activated on a selected substring into the source code under development, or second the programmer inserts the substring in the appropriate field of the tab Identifiers and clicks on the button *Suggest*. The programmer can select the most appropriate identifiers double clicking on it and it will be automatically inserted into the source code. The following figure shows the candidate identifier suggestion by the COCONUT Eclipse plug-in.



```
package org.videoservice.entity;

public class Member {
    class con
```

Substring: con

☐ Whole word

addressContact
confirms
confirmsRequested
contact
contactInformation

1. The Administrator activates the add member function in the terminal of the system and correctly enters his login and password identifying him as an Administrator.

2. The system responds by presenting a form to the Administrator on a terminal screen. The form includes the first and last name, the address, and contact information (phone, email and fax) of the customer. It also displays the membership fee to be paid.

3. The Administrator fills the form and then confirms all the requested form information is correct.

Fig2. The COCONUT view in the Eclipse IDE : Candidate identifier suggestion for source code

IV. APPLYING CODING STYLES

This section describes the proposed approach by using SmartFormatter. The SmartFormatter is a tool that allows programmers to learn coding style rules from existing source code, and apply these rules to the code under development. The format of the source code, style of the source code, and identifiers quality are important aspects that influence program understandings and maintenance. Software comprehensibility is also affected by the availability of proper analysis and design documentation, and the existence of traceability links between source code being written and high-level artefacts.

This tool explanation proposes an Eclipse plugin that learns from existing source code (i) the indentation style, (ii) the conventions used to name different kind of identifiers, and (iii) how and where source code comments are used. After the learning phase, the plugin is able to apply the learned indentation style, to highlight identifiers that violate naming conventions, and the lack of comments in some portions of the source code.

While many tools, including the Eclipse platform, allow for automatic source code indentation, our plugin SmartFormatter differs in that it allows for learning an indentation style — that can be different from the commonly adopted one — from the existing source code and consistently format the code under development.

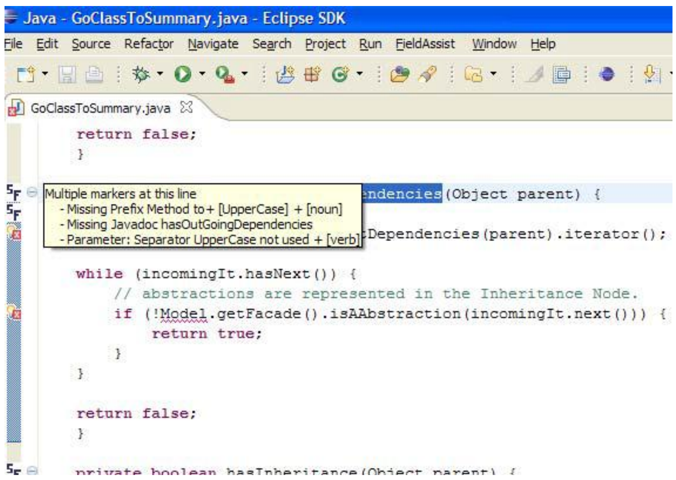
Smart Formatter analyzes source code quality from three different points of view: (i) the indentation style, (ii) the naming convention of identifiers, and (iii) the comment usage and frequency.

The indentation style is learned by analyzing, for each grammar rule, the relative position of each terminal or nonterminal composing the rule with respect to the previous token. The indentation rule is obtained by applying descriptive statistics (average or median) over the collected positions for each instance of the grammar rule.

For the identifiers, the tool learns the style for different identifier categories, i.e., class and interface names, instance variables, method names, parameters, local variables, and constants. The tool attempts to infer, for each category: (i) the identifier prefix, if it is used; (ii) a separator (e.g., camel case or a special character); and (iii) by using the Word-Net1 lexical database, whether the first word of the identifier (excluding the prefix) is a noun, a verb, an adjective, or an adverb. Also in this case, the inference is done by statistically analyzing characteristics of identifiers belonging to the same category. A naming convention, i.e., the presence of a prefix, of a separator or the lexical category of the first word, is learned if it occurs over a given percentage (threshold) of the identifiers belonging to that category.

For the comments, the tool analyzes the comment density and highlight source code files having a comment density below a given percentage of the comment density distribution.

The following figure shows an example of Smart Formatter usage. In this example the tool is able to learn that method parameters use the prefix par, the camel case separator, and the first word should be a verb. For the method names it learns the use of to prefix, the camel case separator and that the first word is a noun. A warning message indicates violations from the rules learned by the tool.


Java - GoClassToSummary.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run FieldAssist Window Help
GoClassToSummary.java

```
    return false;
    }

                                        endencies(Object parent) {
Multiple markers at this line
  - Missing Prefix Method to+ [UpperCase] + [noun]
  - Missing Javadoc hasOutGoingDependencies
  - Parameter: Separator UpperCase not used + [verb]
                                        Dependencies(parent).iterator();

    while (incomingIt.hasNext()) {
        // abstractions are represented in the Inheritance Node.
        if (!Model.getFacade().isAAbstraction(incomingIt.next())) {
            return true;
        }
    }

    return false;
    }

    private boolean hasInheritance(Object parent) {
```

Figure 3. Smart Formatter: violation of naming conventions

V. Conclusion

The paper proposed a novel approach to help developers improve the comprehensibility of source code. In particular, our approach applies styles to the code, that learns style rules from existing source code, and apply these rules to the code under development.

REFERENCES

[1] V.R. Basili, L.C. Briand, and W.L. Melo, “*A Validation of Object-Oriented Design Metrics as Quality Indicators*,” IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751-761, Oct. 1996.

[2] T. Gyimo´thy, R. Ferenc, and I. Siket, “*Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction*,” IEEE Trans. Software Eng., vol. 31, no. 10, pp. 897- 910, Oct. 2005.

[3] M Andrea De Lucia, Senior Member, IEEE, Massimiliano Di Penta, Member, IEEE, and Rocco Oliveto, Member, IEEE ,” *Improving Source Code Lexicon via Traceability and Information Retrieval*” , IEEE Trans. Software Eng., vol. 37, no. 2, March/April 2011

[4] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, “*To CamelCase or under Score*,” Proc. 17th IEEE Int’l Conf. Program Comprehension, 2009.

[5] A. Marcus and J.I. Maletic, “*Recovering Documentation-to- Source-Code Traceability Links Using Latent Semantic Indexing*,” Proc. 25th Int’l Conf. Software Eng., pp. 125-135, 2003.

[6] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, “*Recovering Traceability Links in Software Artefact Management Systems Using Information Retrieval Methods*,” ACM Trans. Software Eng. and Methodology, vol. 16, no. 4, 2007.

[7] Antoniol, G. Casazza, and A. Cimitile, “*Traceability Recovery by Modelling Programmer Behaviour*,” Proc. Seventh Working Conf. Reverse Eng., pp. 240-247, 2000.

[8] S. Haiduc and A. Marcus, “*On the Use of Domain Terms in Source Code*,” Proc. 16th IEEE Int’l Conf. Program Comprehension, pp. 113-122, 2008.