A Visualization Tool Recording Historical Data of Program Comprehension Tasks

Katsuhisa Maruyama
Dept. Computer Science
Ritsumeikan University
Shiga, 525–8577, Japan
maru@cs.ritsumei.ac.jp

Takayuki Omori
Dept. Computer Science
Ritsumeikan University
Shiga, 525–8577, Japan
takayuki@fse.cs.ritsumei.ac.jp

Shinpei Hayashi
Dept. Computer Science
Tokyo Institute of Technology
Tokyo, 152–8552, Japan
hayashi@se.cs.titech.ac.jp

ABSTRACT

Software visualization has become a major technique in program comprehension. Although many tools visualize the structure, behavior, and evolution of a program, they have no concern with how a tool user has understood it. Moreover, they miss the stuff the user has left through trial-and-error processes of his/her program comprehension task. This paper presents a source code visualization tool called *CodeForest*. It uses a forest metaphor to depict source code of Java programs. Each tree represents a class within the program and the collection of trees constitutes a three-dimensional forest. CodeForest helps a user to try a large number of combinations of mapping of software metrics on visual parameters. Moreover, it provides two new types of support: leaving notes that memorize the current understanding and insight along with visualized objects, and automatically recording a user's actions under understanding. The left notes and recorded actions might be used as historical data that would be hints accelerating the current comprehension task.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.8 [**Metrics**]: Product metrics

General Terms

Human Factors

Keywords

Software visualization, software metrics, software reengineering, static analysis, cognitive process

1. INTRODUCTION

Program comprehension is considered to be a task that comprehenders (mainly maintainers) could obtain fresh knowledge about the target programs from their existing understanding [11, 14]. It is a indispensable technique to make software maintenance successful. This is because a maintenance task requires maintainers to obtain the deep knowledge about the target software. For example, the success (or failure) of refactoring would depend on the degrees of knowledge about existing source code of software.

Each comprehender forms a mental model of the target program. It will be done through the cognitive process using his/her previous and new knowledge. To form a plausible mental model, a comprehender tries to correctly capture its structure and behavior. In other words, he/she must grab the facts underlying the program. The use of program comprehension tools would clarify such facts.

A program visualization tool is one of potent candidates that satisfy this requirement [6]. In fact, a number of 2D or 3D visualization approaches have been developed for specific activities of software development and maintenance [5]. Moreover, several results of controlled experiments show a significant improvement in solving a number of program comprehension tasks [8, 12, 16].

There are no questionable points of effective support provided by several visualization tools in solving program comprehension tasks. However, may we say that existing tools provide enough support of their interactions with tool users? Have the tools provided only facts, reasons, and/or evidences as a product that leads to an answer to a question in programming?

To our knowledge, almost all visualization tools for program comprehension limit their functionality to visualization of the structure, behavior, and evolution of a program. The tools only provide different representations captured in various aspects of the program. Unfortunately, they have no concern with how a tool user has understood it and what he/she has left through his/her program comprehension task. If various kinds of facts obtained through visualization are apt to be muddled, a visualization tool should keep track of a user's actions and their visualization results. In this tool, a user himself/herself or other users could easily access such historical data during performing the current task.

This paper presents a visualization tool called *CodeForest*. It uses a forest metaphor to depict source code of Java programs. Each class within the program is represented by a tree and the collection of trees constitutes a three-dimensional forest. The presentation technique of the tool imitates that of CodeCity [15] adopting a city metaphor, and is similar to CodeMetropolis [2] adopting a metropolis metaphor. Similar to CodeCity and CodeMetropolis, CodeForest maps a set of software metrics on the visual parameters of source code. The small difference between these tools is that CodeForest supports more than 10 software metrics and free visual mapping of them on 6 visual parameters by default. In other words, a tool user can try a large number of combinations of visual mapping. This helps wider exploration of visualization. The large difference is that CodeForest provides two types of breadcrumbs. Whenever any questions, conjectures, and answers come into a user's mind, he/she can leave notes that memorize them. More-

ICPC '14, June 2–3, 2014, Hyderabad, India

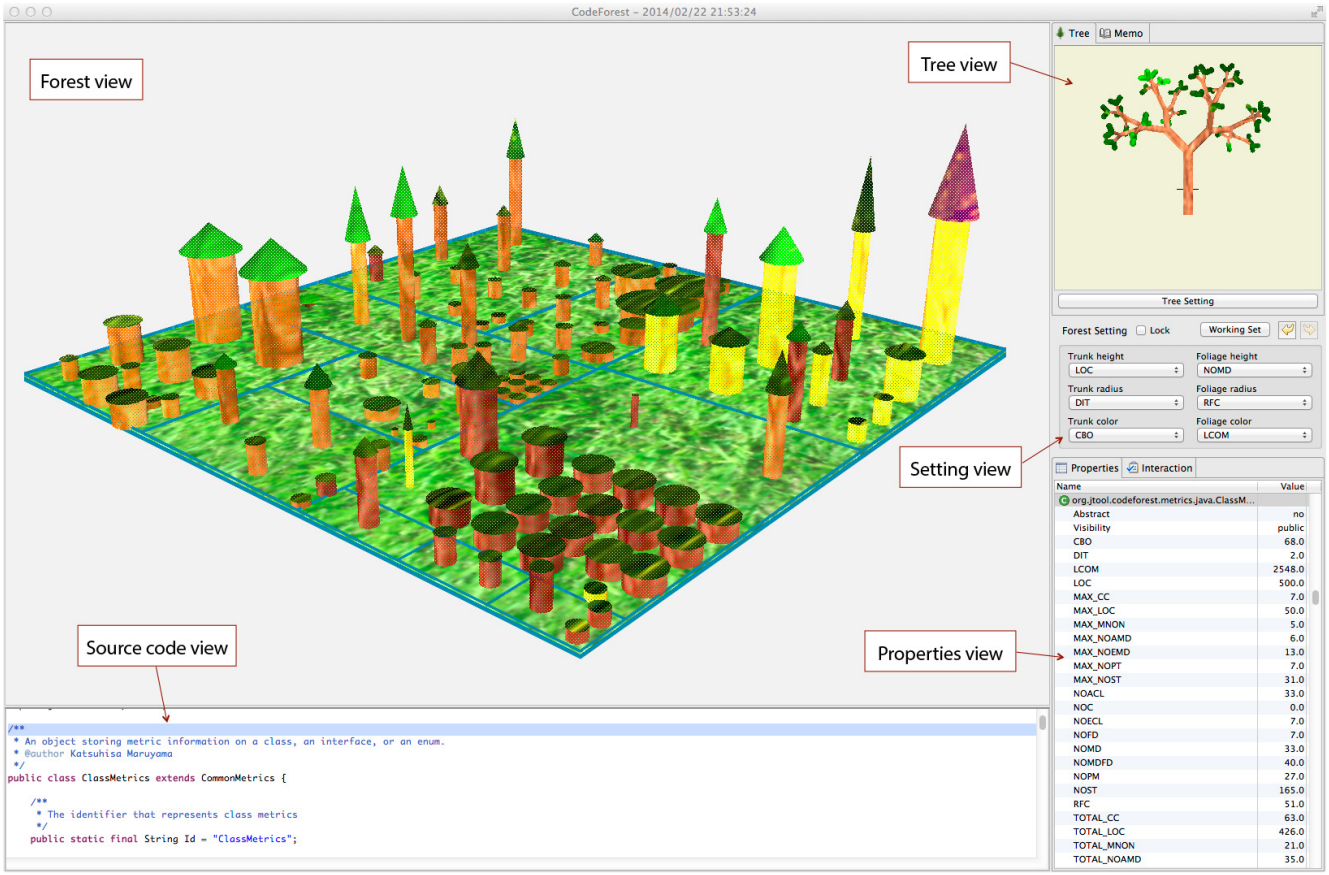

Figure 1: Screenshot of visualization by CodeForest of CodeForest (11 packages and 109 classes).

over, CodeForest automatically records a user's actions such as a change of the setting of visual parameters or a change of the focus of visual objects. The left notes and recorded actions are considered historical information. It helps retrospective of past activities in a user's task of understanding a program. It will be also hints for his/her future understanding or understanding done by others.

The main contribution of this paper is to present a running implementation of a new type of visualization tool that exploits historical information about a user's actions. This tool would show possibility of promoting the usefulness of visualization tools.

2. CODEFOREST

This section first describes how CodeForest visualizes Java source code. Then, it explains what information CodeForest records to support program comprehension.

2.1 Visual Objects and Parameters

Fig. 1 shows a screenshot of visualization by CodeForest. It visualizes Java source code by using two kinds of trees. A tree in a forest view (top left window) provides only information on a class, which is called a *forest tree* hereafter. A tree in a tree view (top right window) provides detailed information on methods of a specified class, which is called a *detailed tree* hereafter. A tool user can switch a detailed tree displayed in the tree view by left-clicking a forest tree displayed in the forest view. A source code view located at left bottom displays source code of a class a user selects in the forest view. A properties view located at right bottom presents information with respect to several properties of source code, which

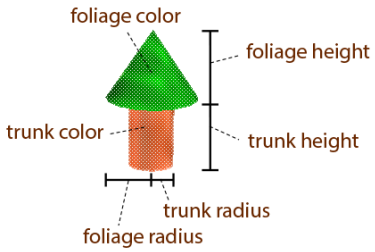

Figure 2: Forest tree with visual parameters.

lists pairs of the name of a software metric and its value. A setting view is located at right middle. In this view, a user can change visual settings that affect the form of a forest tree. In the tree view, he/she can also change visual settings that affect the form of a detailed tree.

Here we will explain the visual parameters in detail. CodeForest provides 6 visual parameters of a forest tree: the trunk height, trunk radius, trunk color, foliage height, foliage radius, and foliage color, as shown in Fig. 2. It also provides 9 visual parameters of a detailed tree: the trunk height, trunk radius, trunk color, branch length, branch radius, branch color, branch number, leaf number, and leaf color. Each branch of a detailed tree represents a method of a class corresponding to the tree. Thus, the value of the branch number is equal to the number of methods within the class. This tree allows a user to change the settings of only the leaf number and color. The trunk height, radius, and color of a detailed tree inherit

Table 1: Software metrics suite

Abbr.	Metrics	Forest (for classes)	Detailed (for methods)
LOC	Number of lines of code	✓	✓
NOST	Number of statements	✓	✓
NOMD	Number of methods	✓	
NOFD	Number of fields	✓	
NOMF	Number of methods and fields	✓	
NOPM	Number of public methods	✓	
NOACL	Number of afferent classes	✓	
NOECL	Number of efferent classes	✓	
CBO	Coupling between objects [4]	✓	
DIT	Depth of inheritance [4]	✓	
NOC	Number of children [4]	✓	
RFC	Response for classes [4]	✓	
WMC	Weighted methods per class [4]	✓	
LCOM	Lack of cohesion methods [4]	✓	
CC	Cyclomatic complexity [9]		✓
MNON	Maximum number of nesting		✓
NOPT	Number of parameters		✓

from those of a forest tree. The branch length and radius are both calculated based on the trunk height and radius, respectively.

In summary, a user can map software metrics to 6 visual parameters of a forest tree and to 2 of a detailed tree. The 17 software metrics shown in Table 1 are totally prepared. Among them, 14 metrics are related to a forest tree and 5 metrics are related to a detailed tree.

2.2 Historical Data in CodeForest

2.2.1 Usage Scene

A user of a visualization tool explores source code by observing its visual representation. In the context of refactoring support, the goal of this exploration is to find bad smells by obtaining comprehensively knowledge of the source code without caring its detailed information. For this, it might be preferable that a visualization tool can provide a variety of visual representations of source code.

In CodeForest, a user has a chance to freely combine 14 software metrics with 6 visual parameters of a forest tree. This means that there are a large number of combinations creating various visual representations as a forest. The user must select one option from among the combinations to obtain a specific visual representation. Here the selected option is called a working set, which is a collection of software metrics mapping on respective visual parameters.

The important point is that one-time selection of a working set never completes program comprehension in most cases. A user selects a working set and interprets its corresponding visual representation. This activity is a cyclic and has been frequently repeated during the process building his/her mental model. For example, the user first assigns the CBO metric to the "trunk color". Then, he/she will pick up some trees based on their tones of color. In Fig. 1, the rightmost tree would be selected since it seems to have high coupling with other classes. He/she would keep this insight in mind and proceeds to a next cycle. Additionally, he/she would memorize the current working set since he/she might come to want to go back the current visual representation after several times of cycles. In short, a user's activity mainly consists of (1) selection of a working set, (2) interpretation of a visual representation, and (3) memorization of the current situation.

2.2.2 Annotation to Visual Objects

Considering capacity limitations of short-term memory of human, free annotation is useful in general. We often write notes instead of memorize the contents of the notes. Such annotation

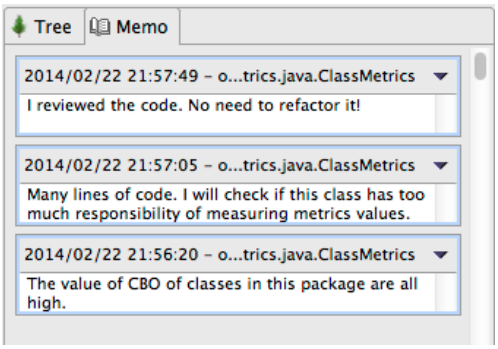

Figure 3: Memo view of CodeForest.

makes retrieval of information at the time from long-term memory more reliable. We believe that free annotation to visual objects in a specific representation is useful in the context of program comprehension. This seems to be plausible because the future version of CodeMetropolis [2] will support of code annotation.

CodeForest enables its user to write notes that memorize information about a specific tree. When he/she selects a visual object by right-clicking it in the forest view, the tree view will be switched to the memo view. This view displays all memos with respect to the selected tree, which are ordered in time (a most recent memo is displayed in the top). Fig. 3 shows the memo view. Three memos were attached to the class `ClassMetrics`, which represent inquiries [7].

2.2.3 Interaction with Visual Representation

We emphasize a user's activity in the cycles of program comprehension tasks as described in Sec. 2.2.1. To support this activity, CodeForest automatically records a user's actions (usage data) of setting visual parameters to reduce the overhead of trial-and-error process. Moreover, it allows him/her to manage working sets of settings of visual parameters. By pushing the "Working Set" button on the setting view shown in Fig. 1, he/she can invoke a dialog that provides functionality of adding a set of the current settings, removing any existing working set, and recalling one of existing working sets. CodeForest also records these addition and removal actions. Fig. 4 shows the interaction view. This view and properties view alternatively appears.

In the interaction view, one line indicates each of a user's actions performed in the past. For example, the line 5 denotes that a user changed software metrics for the visual parameter with respect to "trunk color". The line 8 denotes he/she added a new working set named as "Test 3".

In addition to a user's actions related to settings of visual parameters and working sets, actions related to annotation and undo/redo operations are automatically recorded. The line 16 denotes that a user wrote a new note. The lines 12 and 14 denote that he/she has performed undo and redo operations, respectively. In CodeForest, all changes of settings of visual parameters are stored in the undo list. An undo operation cancels the current settings of visual parameters and recalls their settings used immediately before. The canceled settings are stored in the redo list and can be recalled by a redo operation.

A user can restore any visual representation appearing in the past by double-clicking one of actions except for undo/redo operations. He/she achieves this restoration without troublesome memorization of settings of visual parameters and re-selection of them.

3. RELATED WORK

Investigating a human task for program comprehension, it is ob-

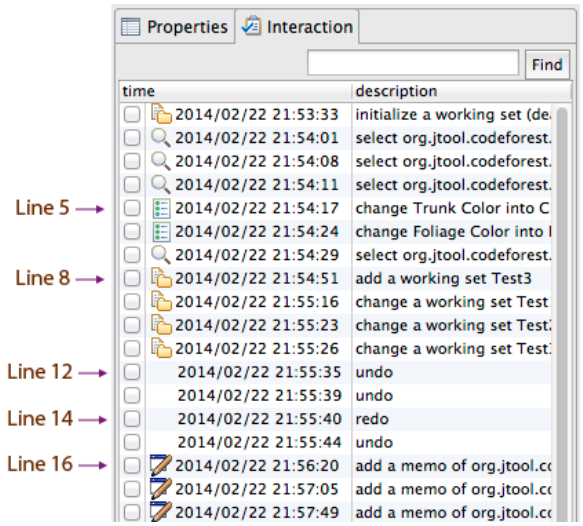

Figure 4: Interaction view of CodeForest.

vious that the task contains a number of iterations of trial-and-error in his/her cognitive process. For example, Brooks claimed that program understanding involves hypothesis generation based on programmer's knowledge and verification process that begins with a primary hypothesis [3]. Moreover, Letovsky found activities called inquiries in programmers' cognitive processes and their repetitions in his empirical study [7]. An inquiry consists of asking a question, conjecturing an answer to the question, and searching to verify the answer. Both the researches reveal that there exists co-evolution of questions about comprehension targets and answers to the questions. Program comprehension process is performed iteratively and incrementally. The changes of understanding level or the results of understanding induce the changes of understanding targets and strategies. Our idea originates from these studies.

The features embedded in CodeForest are not completely novel. Instead, they were inspired by the concepts of the Just in Time Comprehension (JITC) [10], the Integrated Metamodel [13], and the Reverse Engineering Notebook [17]. Singer et al. pointed out that program comprehension tools should provide capabilities to keep track of exploration sessions and allow the navigation of a persistent history. They observed users working on multiple problems over a span of many days and losing information the users had previously found [10]. Mayrhauser and Vans claimed that programmers use an as-needed rather than a systematic strategy for understanding code in an opportunistic understanding process. They also argued that comprehension tools should formulate and keep track of hypothesis, representation of domain knowledge and specialized domain schema, cognition strategies, and analysis of hypothesis failure [13]. Wong claimed that the intended use of the Reverse Engineering Notebook can support continuous program understanding, which enables tool integration with respect to three dimensions: data, control, and presentation [17].

4. CONCLUSION

We expect that CodeForest will facilitate program comprehension tasks using source code visualization. However, we currently have no evidence for the benefits of the use of CodeForest during program comprehension. To check if it can reduce program comprehension effort, we must make several experiments with it. Moreover, the implementation of CodeForest would need improvement of user interface based on the experimental results.

We will also tackle refinement of the currently defined metrics and consideration of a mechanism that enables a user to customize them and add new metrics. The idea of normalizing the scaling factor of each metric, which was implemented in EvoSpace [1], would be useful for the consideration. A mechanism handling memos and interactions of multiple users is a significant issue.

Acknowledgment

The authors would like to thank Daiki Todoroki who engaged the initial implementation of CodeForest. This work was partially sponsored by the Grant-in-Aid for Scientific Research (C) (24500050) from the Japan Society for the Promotion of Science (JSPS).

5. REFERENCES

[1] S. Alam, S. Boccuzzo, R. Wettel, P. Dugerdil, H. Gall, and M. Lanza. EvoSpaces - multi-dimensional navigation spaces for software evolution. In *Human Machine Interaction*, volume 5440 of *LNCS*, pages 167–192. Springer, 2009.

[2] G. Balogh and Á. Beszédes. CodeMetropolis - code visualisation in MineCraft. In *SCAM'13*, pages 136–141, 2013.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *Int'l J. Man-Machine Studies*, 18(6):543–554, 1983.

[4] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng.*, 24(8):629–639, 1998.

[5] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[6] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[7] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987.

[8] J. I. Maletic, A. Marcus, and L. Feng. Source viewer 3D (Sv3D): A framework for software visualization. In *ICSE'03*, pages 812–813, 2003.

[9] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

[10] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON'97*, pages 209–223, 1997.

[11] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[12] M.-A. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On integrating visualization techniques for effective software exploration. In *InfoVis'97*, pages 38–45, 1997.

[13] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tool capabilities. In *CASE'93*, pages 230–239, 1993.

[14] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[15] R. Wettel and M. Lanza. Program comprehension through software habitability. In *ICPC'07*, pages 231–240, 2007.

[16] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *ICSE'11*, pages 551–560, 2011.

[17] K. Wong. *The Reverse Engineering Notebook*. PhD thesis, Univ. of Victoria, 1999.

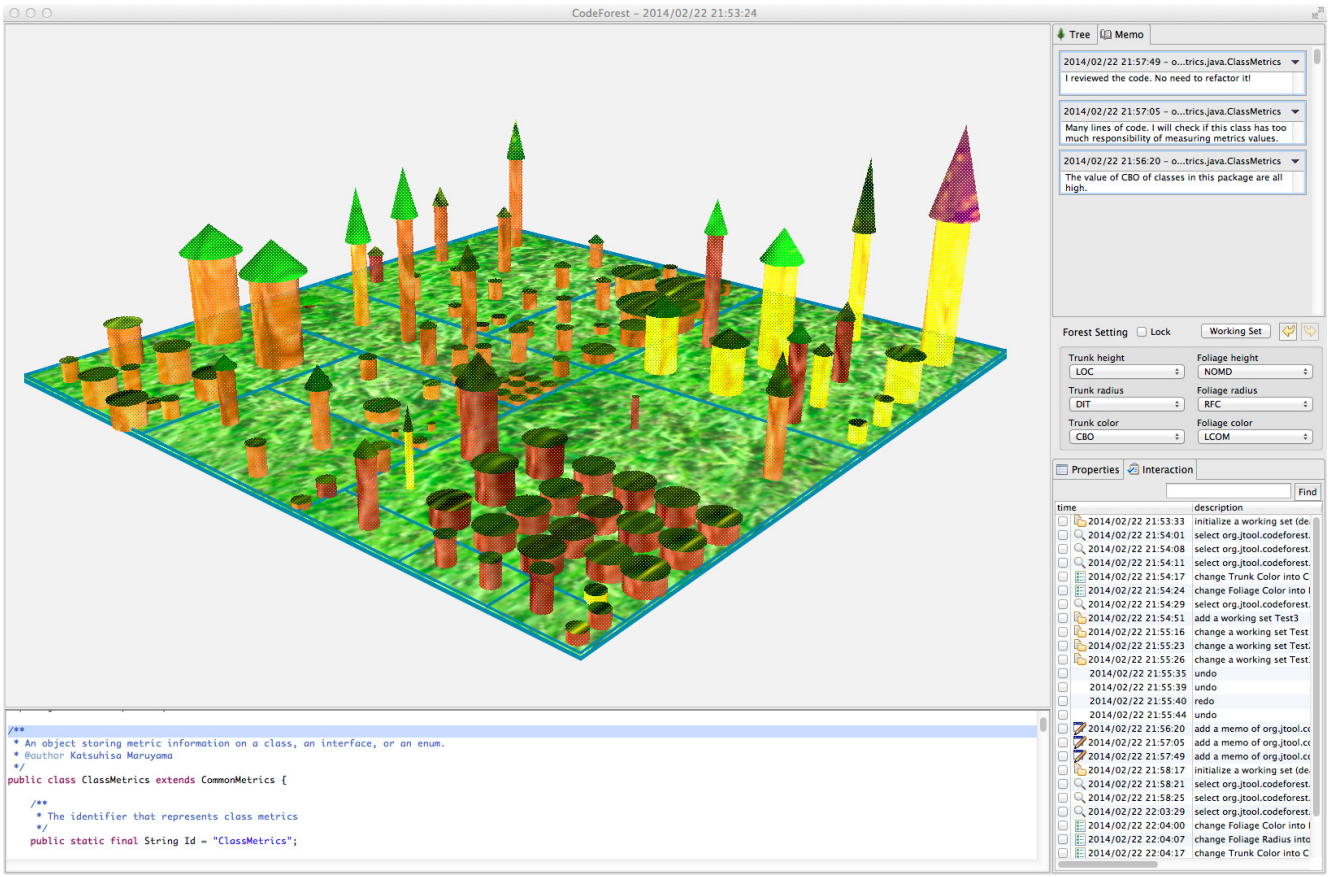

Figure A.1: Screenshot of visualization along with the memo view and the interaction view.

APPENDIX

A. ADDITIONAL SCREENSHOT

Fig. A.1 shows a screenshot of visualization along with the memo view and the interaction view instead of the tree view and the properties view shown in Fig. 1.

B. WEBSITE

The demonstration video of CodeForest can be available from:

`http://www.fse.cs.ritsumei.ac.jp/codeforest/.`

Moreover, the implementation will be available from the above website.

C. REQUIRED MODULES

CodeForest is a plug-in for Eclipse. The following modules are required for its execution.

- Java SE 7
- Eclipse 4.3
  `http://www.eclipse.org/`
- JOGL 2.1.4
  `http://jogamp.org/jogl/www/`
- Java3D 1.6.0 (pre9)
  `http://jogamp.org/deployment/java3d/`
- Jxplatform
  `https://github.com/katsuhisamaruyama/jxplatform`

JOGL is a Java wrapper for the graphics library OpenGL and Java3D provides high-level constructs for creating, rendering, and manipulating a 3D scene graph.

Jxplatform is a tool platform that provides programmer-friendly APIs wrapping AST information provided by Eclipse JDT.