PROPHET: Tool Infrastructure To Support Program Comprehension Experiments

Janet Feigenspan, Norbert Siegmund, Andreas Hasselberg, Markus Köppen
University of Magdeburg, Germany
{feigensp, nsiegmun}@ovgu.de, {hasselbe, mkoeppen}@st.ovgu.de

***Abstract*—Program comprehension is an important human factor in software engineering. To evaluate and improve it, controlled experiments are usually necessary, which, however, require a lot of effort. To support researchers in planning and conducting program-comprehension experiments, we developed the extensible tool PROPHET. It can fulfill the most common requirements of program-comprehension experiments published in the last five years. Furthermore, PROPHET provides an extensible architecture, so that new requirements can be easily implemented.**

***Keywords*-program comprehension; experiments; tool support; prophet**

I. Introduction

Since the development of the first programmable computers, program comprehension is an important human factor in the computer-science domain. Understanding a program is an important task for maintenance developers and it contributes to about 50–60 % to a developer's activity [13]. Hence, programs that are easy to comprehend can save time of software development. Furthermore, since maintaining software contributes to up to 70 % of cost of software development [1], understandable programs can save considerable amounts of costs.

Numerous languages and paradigms were developed to improve comprehension of source code, for example, Assembler languages [9], Ada [2], or Java [8]. To analyze the effects of new programming techniques on program comprehension, controlled experiments are necessary, because program comprehension is an internal cognitive process [5]. Despite the importance of empirical evaluation of program comprehension, controlled experiments are still conducted reluctantly compared to other disciplines [11], [14]. In a recent study, Sjoberg et al. concluded that only about 1.9 % papers published in twelve leading conferences and journals in the software-engineering domain reported controlled experiments [10].

Tichy names several reasons for the low amount of empirical research, for example that it may slow down progress, that technologies may change too fast, or that it is hard to publish results of empirical studies in the software-engineering community [12]. One main argument is that experiments are too time consuming and costly. For example, experimental material has to be designed, subjects recruited, and data analyzed. Furthermore, to generalize results of one experiment, it should be replicated [4]. Hence, when designing experiments, experimenters have to ensure that the experiment is repeatable and that the results are verifiable [3]. Verifiability and replicability are important to support other researchers in evaluating the thoroughness and correctness of the experimental design.

Program-comprehension experiments usually require tool support, for example, to visualize source code and other implementation artifacts or to log the behavior of subjects. Despite of such common functional requirements, different experimenters often develop their own tool instead of reusing already existing ones (e.g., [7]). However, this practice makes verifying and replicating experiments difficult, because the underlying tool infrastructure may not be available or creating similar conditions for replication is not possible.

In this paper, we present a tool called PROPHET to support program-comprehension experiments. Our aim is to provide a tool infrastructure that can be used and extended by other researchers to design and conduct experiments. This way, experimenters can concentrate on experimental design rather than developing a tool to conduct an experiment. We developed PROPHET based on a literature review of controlled experiments measuring program comprehension published in the last five years. PROPHET is highly customizable and can support most of the reviewed experiments. Furthermore, we support replication of experiments, because settings can easily be made available for other researchers. Additionally, PROPHET is highly extensible due to a plug-in structure, such that requirements currently not met can be integrated.[1]

II. Literature Review

In this section, we summarize the literature review we conducted to identify commonly used features in program-comprehension experiments. We analyzed papers published between 2006 and 2010 that report controlled experiments with human subjects from the *Journal of Empirical Software Engineering (ESE)*, *International Conference on Program Comprehension (ICPC)*, and *International Conference on Software Engineering (ICSE)*. We selected the journal and conferences, because they are the leading publication platforms in (empirical) software engineering and program comprehension. The literature review is work in progress. In future work, we plan to extend the review to more journals and conferences and to more issues. Nevertheless, we received a good impression of common features of program-comprehension experiments.

So far, we identified the following features: source-code viewer, measurement of time, presenting tasks/questionnaires, logging, and external tool usage. First, in all papers we reviewed, source code was presented to subjects, which is inherent for program-comprehension experiments. Second,

[1]PROPHET and some additional material is available at http://fosd.net.

Component	ICSE	ICPC	ESE	Total (%)
Source code viewing	17	17	18	52 (100 %)
Measurement of time	10	12	11	33 (63 %)
Presenting tasks/questionnaires	17	17	18	52 (100 %)
Logging	6	6	6	18 (35 %)
External tool	5	7	6	18 (35 %)

TABLE I
OVERVIEW OF FEATURES IN PROGRAM-COMPREHENSION EXPERIMENTS.

time was often measured, either because of time constraints for a session, or to analyze how long subjects needed to complete an experiment. Third, tasks were often used to measure program comprehension. In some papers, we also found that questionnaires were used, for example to assess the opinion of subjects. Fourth, in some experiments, the actions of subjects were logged and used to analyze the behavior. Last, many experiments used external tools, which means that tools beside the actual experimentation environment are used.

In Table I, we present an overview of the amount of experiments having the requirements we identified so far. All experiments need to present source code to subjects and used questionnaires and/or presented tasks. Almost two third of the experiments measured the time. Furthermore, a third of the experiments needed a logging feature. In most experiments, authors implemented their own tool infrastructure or used existing tools, such as Eclipse with or without extensions. Some experiments were even paper based. However, this provides two drawbacks: First, it is hardly legitimate to compare the results of an experiment that used Eclipse to an experiment that was paper based. With Eclipse, experimenters can use a lot of tool support, such as search functions or auto completion, or compiling and running source code, which is not possible if subjects work with a sheet of paper.

Second, replicating experiments is difficult. For replication, the same or similar conditions have to be created. This can be very tedious, especially when experiments are described with a strict space limit, such as it exists for conferences. Even if a lot of information for replication is presented, replicating them is not necessarily possible [6]. To support comparability and replication of experiments, we provide a tool infrastructure, which we present next.

III. TOOL INFRASTRUCTURE

In this section, we give an overview of our tool PROPHET. We implemented it as a plug in system, such that new features can easily be implemented and added to PROPHET. Our aim is to support planning and conducting experiments. Therefore, we divide our tool in two main parts: *Experimenter View* and *Subject View*. In the following, we describe each part of the tool in detail.

A. Experimenter View

In the *Experimenter View*, we provide a user interface for experimenters that allows them to customize the experimental setting according to their needs. In Fig. 1, we show a screenshot of the *Experimenter View*. On the left side, we provide an overview of existing tasks. For each task, we have four different tabs that provide customization options. The content of the left tab *Editor* allows an experimenter to enter text, for example, for a task description. The tab *Preview* in Fig. 1 shows a preview of how subjects see the description of a task. The tab *Preferences* encapsulates numerous customization options to present source code or tasks to subjects and to activate plug ins. The last tab *Notes* allows an experimenter to save notes for a certain task. In the next paragraphs, we take a closer look at the content of each tab.

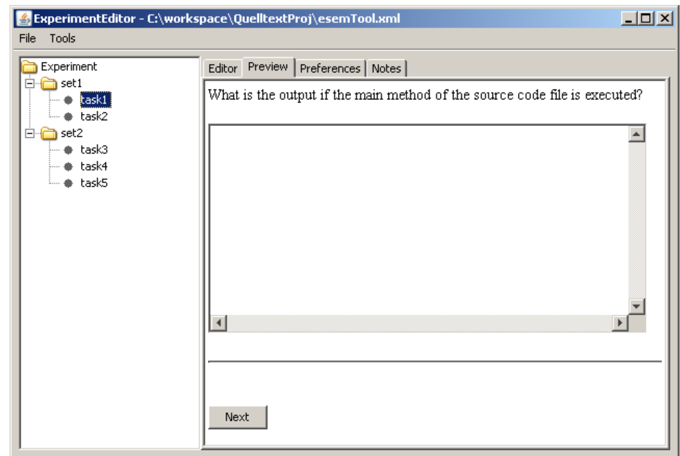

Fig. 1. Screenshot of the *Experimenter View* of PROPHET.

First, we have an overview of the experiment and all defined tasks on the left side (referred to as *Package Explorer*). We can define different sets of task and apply the same customizations to it. For example, in Fig. 1, we have two sets of tasks, set1 and set2. In the first set, we defined task1 and task2, and in the second set, task3, task4, and task5. For both sets, we can specify different settings. To this end, we look at the content of the tabs in the main window of the *Experimenter View*.

In Fig. 1, we display the *Preview* tab, in which we show how a description is displayed in the *Subject View*. Furthermore, in the right most tab *Notes*, we can write notes for tasks in simple text format (e.g., deviations that occurred or for researchers who replicate the experiment).

In Fig. 2, we show the content of the *Editor* tab. It allows experimenters to define descriptions for tasks. The tab contains an editor, in which the descriptions of tasks can be defined as HTML code. For convenience, we provide templates for often used HTML tags, such as font type and forms. Furthermore, we can define macros to support creating task descriptions.

In the tab *Preferences*, shown in Fig. 3, the experimenter can customize the experimental setting. For example, if we check the box *Activate code viewer*, we have numerous options to customize how subjects see source code. For example, we can:

(1) define a folder of which the contents are shown in the *Package Explorer* in the *Subjects View* (Section III-B),
(2) define a file that is displayed when a task begins,
(3) choose whether source code is editable by subjects,
(4) choose what behavior of subjects we log,
(5) choose whether subjects can use a search feature.

Furthermore, we can define a time out at which the ex-

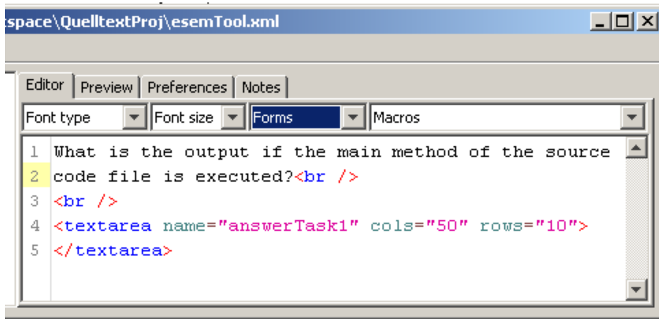

Fig. 2. Screenshot of the *Editor* tab.

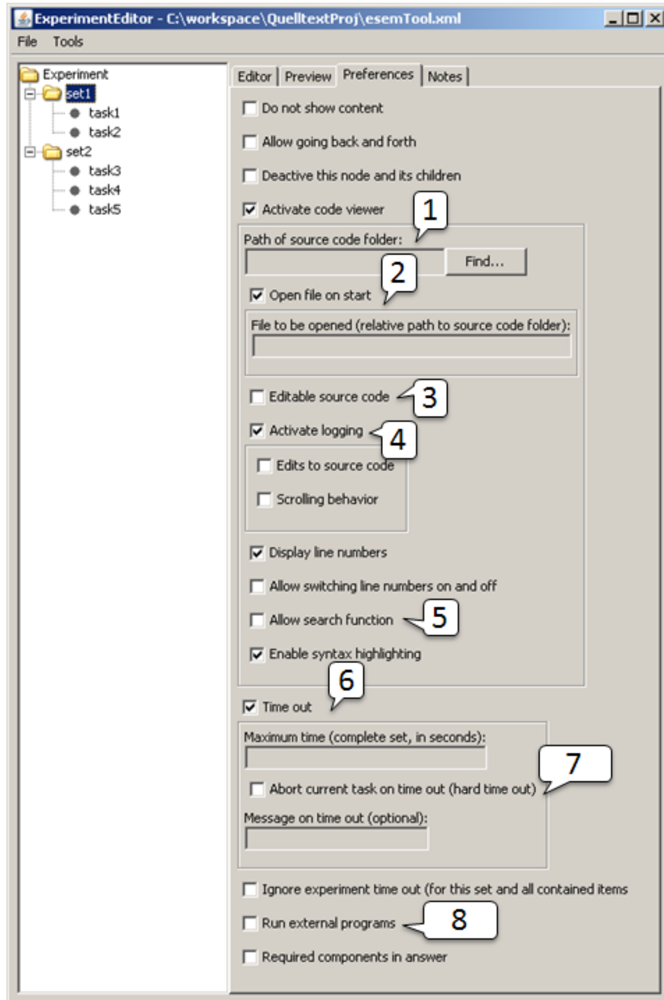

Fig. 3. Screenshot of the *Preferences* tab. The numbers refer to options we explain in detail.

periment is aborted (6). We can define whether subjects are allowed to finish the currently displayed task (soft time out) or not (hard time out, 7). Moreover, we can define whether we want to start external programs, like a web browser (8).

In addition to defining settings for a set of tasks, we can define settings for the complete experiment. In Fig. 4, we show a screenshot of the currently implemented options. The most interesting feature here is to send an e-mail. If this option is selected, the logging files will be automatically zipped and

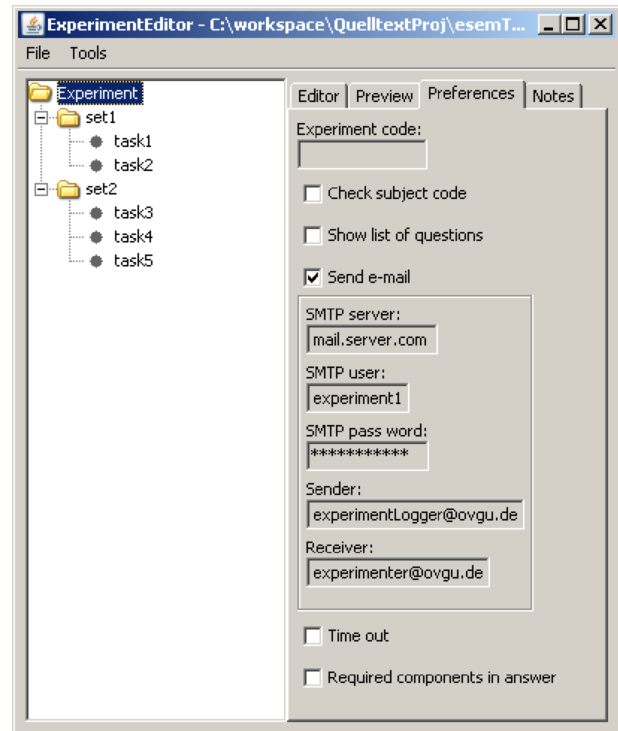

Fig. 4. Screenshot of *Preferences* tab for the complete Experiment.

sent via e-mail at the end of the experiment, without requiring any interactions from subjects. The sender and receiver mail addresses can be specified by the experimenter. All settings for an experiment are stored as XML file. This way, the experiment can be easily shared with other researchers.

In the menu item *File*, we have two export options. The first one allows exporting all questions defined in an experiment to an HTML file, for example, to present them at a website for other researchers. The second option allows exporting the answers and logging data of subjects to CSV files, such that standard statistic tools (e.g., R and SPSS) can read the data.

B. Subject View

Our tool creates a *Subject View*, with which subjects work during the experiment. Typically, it is divided into a source-code viewer and a task viewer. In the source-code viewer, we present source code to subjects with the specifications defined in the *Preferences* tab of the *Experimenter View*. We show an example of the source-code viewer in Fig. 5.

In the task viewer, we present the tasks to subjects. It is similar to the *Experimenter View* in Figure 1. It provides an overview of the tasks, the task description, the text field for the answer, a button to submit an answer, and the time that has passed since the beginning of the experiment, as well as the time for a set of tasks.

IV. Evaluation

In this section, we discuss how PROPHET supports the features of program-comprehension experiments identified in Section II.

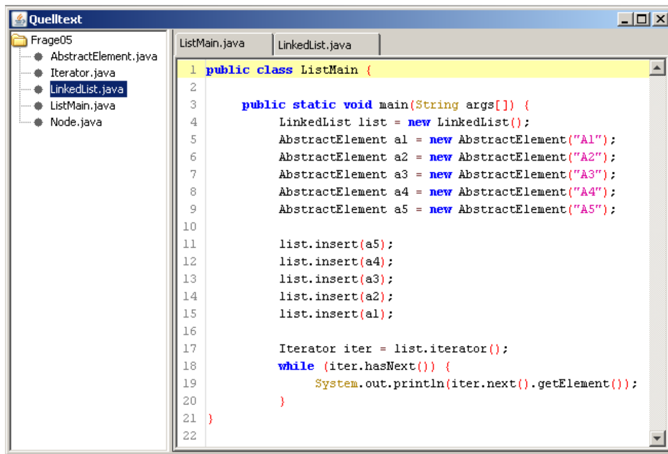

Fig. 5. Source-code viewer in our experiment.

1) Source-code viewer: Our tool provides a source-code viewer with several options to customize how subjects see source code. We can customize for each task how a single source-code file is displayed to subjects. For example, we can turn line numbers or syntax highlighting on or off. We can display a *Package Explorer* that shows the specified contents of a project. This way, we provide the basic component of program-comprehension experiments, which can be customized according to the experimenter's needs.

2) Measurement of time: In the *Experimenter View*, we have several customizing options to measure the time and set a time limit at which the experiment automatically ends. Hence, we can support all experiments that need to measure the response times of subjects and that have a time constraint.

3) Presenting tasks/questionnaires: For presenting tasks to subjects, we can write HTML code for each task in an editor. For convenience, we support templates and macros for commonly used HTML tags. This way, we support all experiments that present tasks or questionnaires to subjects.

4) Logging: Our tool is able to log each action of subjects with a time stamp, for example, opening files or entering an answer. This way, we enable experimenters to conduct a detailed analysis of subjects' behavior during the experiment. Hence, we can support most experiments in which subjects should complete a questionnaire or tasks and in which the behavior of subjects during the experiment needs to be analyzed.

5) External tools: In our tool, we can specify a set of tools that subjects are allowed to use. Tools such as a web browser can simply be specified in the *Preferences* tab of PROPHET. However, in the current version, we cannot specify a compiler or interpreter to compile or execute source code, which might be necessary for program-comprehension experiments. Nevertheless, the plug-in architecture of our tool allows us (or other researchers) to extend our tool to call a compiler or interpreter.

Hence, we can support the requirements we identified so far. To further evaluate the usefulness of PROPHET, we conducted a program-comprehension experiment, in which we presented tasks, source code, and questionnaires to subjects. Furthermore, we implemented several plug ins to evaluate its extensibility mechanism. So far, we did not encounter any problems; PROPHET fully supported our experiment and we could implement new plug ins without changing the base code. However, to have a more objective view about PROPHET, we encourage other researchers to use PROPHET for their experiments and report their experience.

V. Conclusion

Program comprehension is an important human factor in software engineering. To support the conduction of program-comprehension experiments, we developed a tool called PROPHET. It supports most of the identified features of program-comprehension experiments. It is highly customizable and its plug-in structure allows researchers to implement new features without changing the existing source code of PROPHET. Hence, we provide a suitable tool infrastructure for program-comprehension experiments. This way, we help researchers to plan and conduct their experiments. Furthermore, we provide a good base for replication, since all experimental settings are stored and can be reused.

In future work, we plan to extend our literature review. This way, we have a more exhaustive overview of features for program-comprehension experiments and can add commonly used features to PROPHET. Furthermore, we plan to conduct more experiments based on PROPHET to confirm its usefulness. We also would like to encourage other researchers to use PROPHET and report how it helps them.

Acknowledgment

Feigenspan's and Siegmund's work is supported by BMBF project Nb. 01IM10002B.

References

[1] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
[2] W. Carlson, L. Druffel, D. Fisher, and W. Whitaker. Introducing Ada. In *ACM '80: Proceedings of the ACM 1980 Annual Conference*, pages 263–271. ACM Press, 1980.
[3] C. Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.
[4] N. Juristo and S. Vegas. The Role of Non-exact Replications in Software Engineering Experiments. *Empirical Softw. Eng.*, 2010. Online first.
[5] J. Koenemann and S. Robertson. Expert Problem Solving Strategies for Program Comprehension. In *Proc. Conf. Human Factors in Computing Systems (CHI)*, pages 125–130. ACM Press, 1991.
[6] J. Lung, J. Aranda, and S. Easterbrook. On the Difficulty of Replicating Human Subjects Studies in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 191–200. ACM Press, 2008.
[7] N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, and A. van Deursen. Connecting Traces: Understanding Client-Server Interactions in Ajax Applications. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 216–225. IEEE CS, 2010.
[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
[9] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, 1992.
[10] D. Sjoberg, J. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.
[11] W. Tichy, P. Lukowicz, L. Prechelt, and E. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, 1995.
[12] W. F. Tichy. Should Computer Scientists Experiment More? *Computer*, 31(5):32–40, 1998.
[13] A. von Mayrhauser, A. Vans, and A. Howe. Program Understanding Behaviour during Enhancement of Large-scale Software. *Journal of Software Maintenance: Research and Practice*, 9(5):299–327, 1997.
[14] M. Zelkowitz and D. Wallace. Experimental Validation in Software Engineering. *Information and Software Technology*, 39(11):735–743, 1997.