This is the authors' version and intended for personal use only. Another other use may violate the copyright.

Supporting Comprehension Experiments with Human Subjects

Janet Feigenspan, Norbert Siegmund
University of Magdeburg

I. Introduction

In software engineering, researchers increasingly conduct controlled experiments with human subjects [2]. Often, the comprehensibility of new methods, tools, and language mechanisms is evaluated [1], [6], [9].

However, conducting experiments is often tedious, because confounding parameters can bias the result and need to be controlled [4]. This is one of the reasons why there is still too little empirical research in the software-engineering community [11]. Furthermore, replication is necessary to confirm and generalize results of experiments [10]. To this end, experimental designs need to be repeatable and verifiable [8].

To support experimenters in designing and replicating experiments, we developed a tool called PROPHET (PRogram cOmPreHension Experiment Tool). It supports experimenters in presenting experimental material, such as source code, images, tasks, and questionnaires, based on HTML. It offers numerous customizing options, such as allowing/hiding search functionality or making the source code editable/non-editable.

We based the design of PROPHET on a literature review, in which we evaluated features used in other experiments. To this end, we reviewed papers of one journal and two major conferences in this domain.[1] As a result, we extracted common required functionalities a tool should provide, which we summarize in Table I. All selected papers describe experiments that presented source code or similar material to subjects as well as the task and questionnaires. In 63 % of the papers, either the experimenter or the subject measured the time to perform a task. In about a third of the papers, authors logged the behavior of subjects (e.g., to evaluate their navigation behavior) and used an external tool (e.g., a Browser to let subjects search in APIs). We implemented these identified features in PROPHET, which is available at http://fosd.net/prophet

II. PROPHET

PROPHET provides two different views, one for the experimenter to specify the experimental setup, and one for the subjects to view the material and perform the tasks.

[1] *Journal of Empirical Software Engineering (ESE)*, *International Conference on Program Comprehension (ICPC)*, and *International Conference on Software Engineering (ICSE)* of the years 2006 to 2010 as leading platform in their field. We plan to include other journals and conferences in the future.

TABLE I
Overview of features in program-comprehension experiments.

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 %)

a) Experimenter View: Each experiment has global settings. In Fig. 3, we show the preferences for a complete experiment, where we can select feature *Send e-mail*. It zips each subject's logged data of a completed experiment and automatically sends them from the *Sender* address to the *Receiver* address. If sending fails, PROPHET produces a message.

Furthermore, we can set a time out for the complete experiment. When time has run out, the experiment ends, sends the data via mail (if selected), and notifies the subject.

To define experimental tasks, we use HTML (Fig. 4). For efficiency, we provide common formatting options in drop down lists and allow experimenters to define their own macros. The tab *Preview*, depicted in Fig. 5, shows how the HTML code is presented to subjects. Additionally, the tab *Notes* allows experimenters to take notes for a task, e.g., to protocol deviations.

To customize a single task, PROPHET provides numerous options, shown in Fig. 6. For better overview, we labeled the important features and refer to the labels in the text. When we select the code viewer (checkbox *Activate code viewer)*, we have numerous customizing options, for example:

(1) define a folder of which the contents are shown in the *Subject View* (Section II-0b),
(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, and
(5) choose whether subjects can use a search feature.

In addition, we can define a time out (6) for a task and specify whether subjects are allowed to complete the task (7). Moreover, we can specify additional programs that a subject is allowed to use (8).

All settings of the experimenter view are stored in an XML file to support replicability. This way, researchers replicating experiments can use the XML file of the original experiment.

This is the authors' version and intended for personal use only. Another other use may violate the copyright.

```
1  public interface PluginInterface {
2
3  //Delivers settings components shown in the settings tab of the experiment editor.
4  public SettingsComponentDescription
       getSettingsComponentDescription(QuestionTreeNode
       node);
5
6  //Called once when the experiment viewer is initialized.
7  public void experimentViewerRun(ExperimentViewer
       experimentViewer);
8
9  //If any plugin denies the currentNode to be entered (e.g., because of a timeout),
       it will be skipped.
10 public boolean denyEnterNode(QuestionTreeNode node);
11
12 //The node entered
13 public void enterNode(QuestionTreeNode node);
14
15 //A message shown to the subject to indicate what needs to be done to accept
       finishing this node (e.g. enter a needed answer)
16 public String denyNextNode(QuestionTreeNode currentNode);
17
18 //The node to be exited
19 public void exitNode(QuestionTreeNode node);
20
21 //A unique name for the plugin.
22 public String getKey();
23
24 //A message shown to the subject at experiment's end
25 public String finishExperiment();}
```

Fig. 1. Source code of the interface used to define new plug-ins.

b) Subject View: Subjects see the tasks as shown in Fig. 5 with the elapsed time at the bottom for the complete experiment and for each task. On the left side, subjects see the folder and contents as specified by the experimenter. Source code is presented in an extra window (the code viewer, Fig. 7), which opens when subjects start a task (if specified) and which is customized according to the previously defined settings.

c) Customizability and Extensibility: To provide support beyond the implemented features, we designed PROPHET to be extensible by using plug-ins [3]. For example, syntax highlighting or search functionality are implemented as plug-ins. In Figure 1, we show the plug-in interface. A plug-in defines which customization options it provides. In Line 4 of Figure 1, the type `SettingsComponentDescription` is used to describe possible settings of the plug-in. For example, consider a plug-in that allows an experimenter to deactivate a certain task (e.g., because we use different groups of subjects working on different tasks). In Figure 2, we show parts of the source code of such a plug-in. In Line 2, we override the interface method of Line 4 of Figure 1 to introduce a checkbox in the *Experimenter View* labeled *Deactivate current and all subnodes*.

Another extension point of our tool represents a plug-in itself. That is, a plug-in can use or define other plug-ins. For example, an experimenter can extend existing plug-ins to tailor them for a specific scenario. We already developed two plug-ins that make use of this concept. We refined the data recorder plug-in with sub plug-ins to tailor the logging behavior of our tool. We extended the source-code viewer to increase customizability with the following sub plug-ins: *EditAndSave*, *LineNumbers*, *OpenedFromStart*, *Recorder*, *SearchBar*, and *SyntaxHighlighting*. The source-code viewer defines an inter-

```
1  class InactivityPlugin implements PluginInterface{
2  public SettingsComponentDescription
       getSettingsComponentDescription(
3        QuestionTreeNode node) {
4    if (!node.isExperiment()) {
5      return new SettingsComponentDescription(
           SettingsCheckBox.class, KEY,
6            "Deactivate current and all subnodes.");
7    } else {
8      return null;
9    }
10 }
11 ..
12 public boolean denyEnterNode(QuestionTreeNode node) {
13   return Boolean.parseBoolean(node.getAttributeValue(KEY));
14 }}
```

Fig. 2. Source code of a plug-in to deactivate certain tasks.

face that a sub plug-in has to implement. For example, the interface has a method to delegate customization options for experimenters to the plug-in structure of our tool, such as showing the option whether line numbers are visible in the viewer.

So far, we used PROPHET successfully in several experiments (e.g., [5], [7]). To have more data for a more thorough evaluation of our tool, we encourage other researchers to use our tool and send us their experiences.

Acknowledgments

Feigenspan's and Siegmund's work is supported by BMBF project 01IM10002B. Thanks to Jana Schumann, Andreas Hasselberg, and Markus Köppen for their support in the literature study and the implementation.

References

[1] E. Arisholm. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Trans. Softw. Eng.*, 33(2):65–86, 2007.

[2] R. P. Buse, C. Sadowski, and W. Weimer. Benefits and Barriers of User Evaluation in Software Engineering Research. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 643–656. ACM Press, 2011.

[3] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley, second edition, 2006.

[4] J. Feigenspan et al. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *Proc. Int'l Workshop on Feature-Oriented Software Development*, pages 55–62. ACM Press, 2009.

[5] J. Feigenspan et al. Exploring Software Measures to Assess Program Comprehension. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE CS, 2011. paper 3.

[6] J. Feigenspan et al. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75. Institution of Engineering and Technology, 2011.

[7] J. Feigenspan et al. Measuring Programming Experience. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE CS, 2012. To appear.

[8] C. Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.

[9] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter. Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 156–167. IEEE CS, 2009.

[10] N. Juristo and S. Vegas. The Role of Non-exact Replications in Software Engineering Experiments. *Empirical Softw. Eng.*, 2010. Online first.

[11] W. F. Tichy. Should Computer Scientists Experiment More? *Computer*, 31(5):32–40, 1998.

This is the authors' version and intended for personal use only. Another other use may violate the copyright.

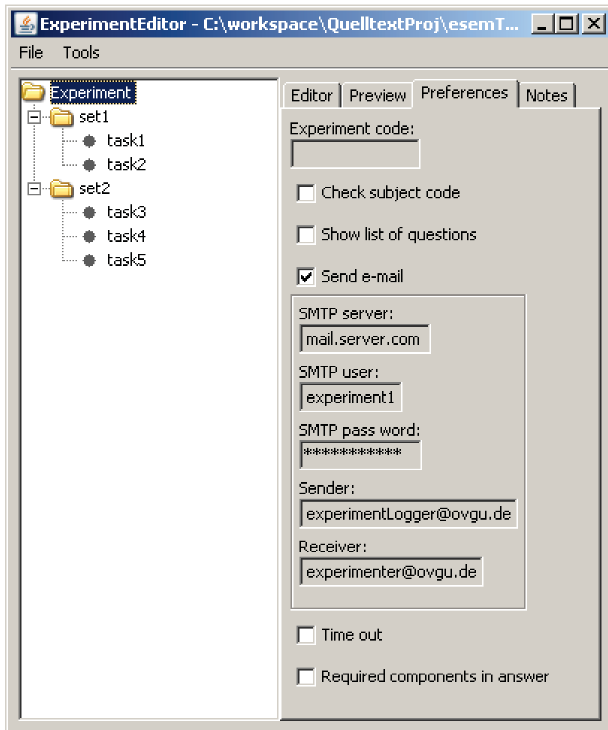

Fig. 3. Screenshot of *Preferences* tab for the complete experiment.

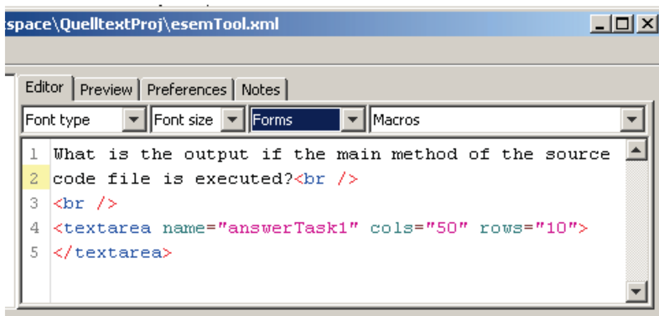

Fig. 4. Screenshot of the *Editor* tab.

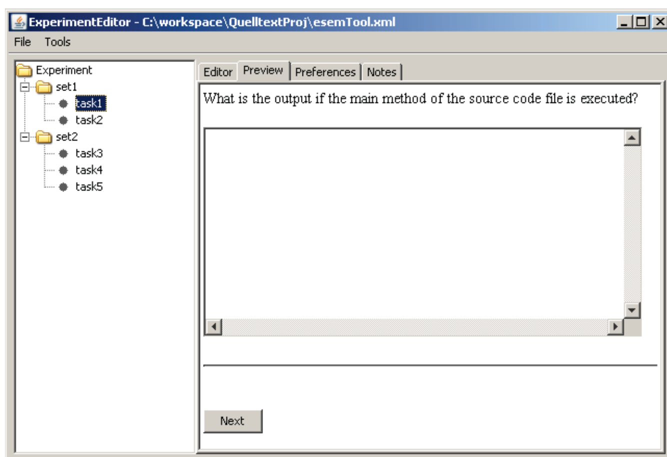

Fig. 5. Screenshot of the *Experimenter View* of PROPHET.

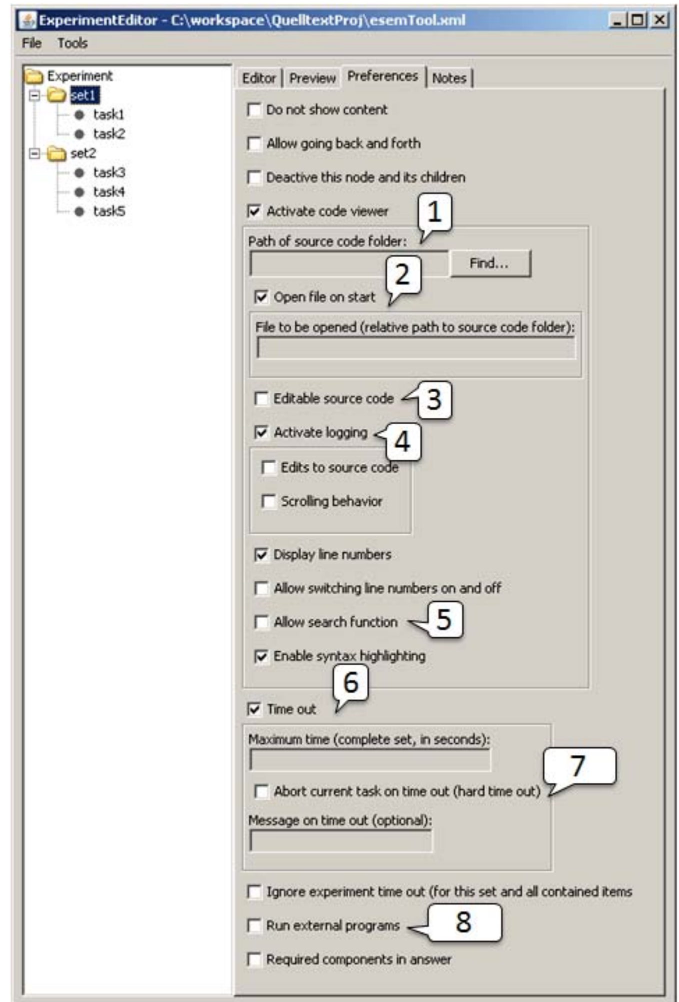

Fig. 6. Screenshot of the *Preferences* tab. The numbers refer to options we explain in detail.

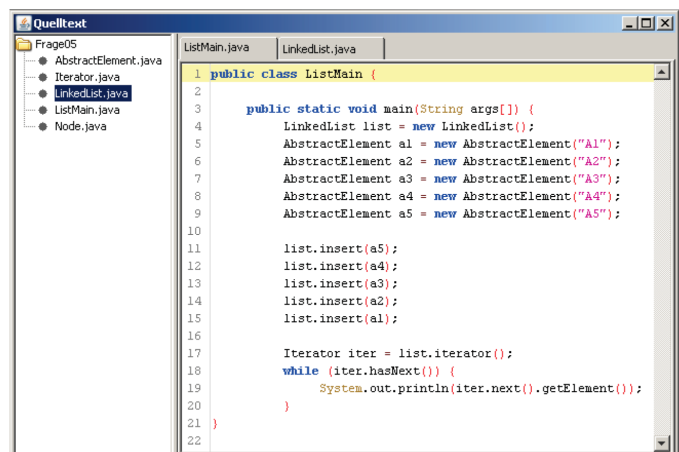

Fig. 7. Source-code viewer in our experiment.