Enabling Code Adaptation for Non-Programmers

Paul Gross
Washington University in St. Louis
grosspa@cse.wustl.edu

1. Introduction

Online code repositories' growth presents new learning opportunities for end-users. Examples of various program features, both novice and advanced, are present for anyone to explore. However, the utility of these examples varies by an end-user's experience level. For instance, a user may observe a program output feature they want (e.g., a button rollover) in an unfamiliar program. An inexperienced end-user may be unable to *find* this code of interest and *extract* it for their use. Further, if a user can find the responsible code and extract it, they may be unable to successfully *integrate* the code into their work due to differences in code structure, dependencies they do not recognize, or other possible issues. End-users potential difficulty with using these examples, particularly those with no programming experience, or non-programmers, may inhibit their learning from these highly available and varied programming resources.

To help end-users utilize these resources we propose a software solution to assist in this *code adaptation* process. Our goal is to support this process without requiring end-user programming experience by leveraging the intuitive connection between the timing of graphical program output and executing code.

2. Related Work

Software Adaptation [1] studies component reusability at many functional levels and is more concerned with choosing software components than users reading and selecting program code. Ko et al. [5] researched how experienced developers find and use information in code maintenance tasks which includes finding responsible code. Similarly the Whyline [4] and WYSIWYT [6] presented different interactions for end-user fault localization. Sato, Shizuki, and Tanaka [7] proposed ORCA to relate transitioning GUI elements with executing code in a single treemap view.

3. Methods

The initial target programming environment for our software solution is Looking Glass, the next version of Storytelling Alice [3]. Looking Glass allows users to create interactive 3D animated stories by writing programs that invoke methods (e.g., walk, say) on objects (e.g., fairy, person). Although Looking Glass is neither a general purpose IDE nor language, it does make program code approachable for our non-programmer target group. Looking Glass has graphical output, intuitive code, and syntax error prevention.

The code adaptation process begins with *finding* code responsible for observed graphical output functionality, thus we began with this problem. To assist non-programmers in finding responsible code requires discovering what information non-programmers naturally use in their search, what natural processes they employ with that information, and where these processes fail.

We explored 14 non-programmers' natural search processes when asked to find code responsible for an observable program output. Subjects successfully identified the correct code on 41% of tasks. We examined what types of searches types users employed (e.g., keyword scanning, narrowing search space through timing) and what barriers inhibited their performance (e.g., misinterpreting methods, ignoring execution flow changes, not fully navigating code) [2].

To address the most prominent user struggles in finding code we developed Hastings. Hastings allows users to explore a program's execution after it runs with an execution time slider, a screen shot of the graphical output at the current time, a list of methods executing at the current time, and an annotated code view showing what lines of code executed and in what order. Users can correlate the output view to what code was executing. This correlation assists them in understanding what a method did and how execution flow changed. The code views afford code navigation features helping users find all relevant code.

To evaluate Hastings effectiveness we will conduct another user study similar to before. This study instance will have three user groups: without search support, with a debugger, and with our software. We compare with a debugger because it is a commonly available tool that can relate lines of code to output functionality timing through breakpoints and stepping. We will evaluate groups' performance by the percentage of correct answers and average time taken per task. The result will determine if our software does improve users' performance in finding responsible code and if it is better than a debugger for these tasks.

Once we have successful software for finding code of interest we can explore the next adaptation process step of *extracting* code and reusing it elsewhere. We will construct a prototype system allowing users to specify the code of interest's beginning and ending, "copy" it out, and "paste" it in other programs. We will conduct a user study to evaluate the software's usability for non-programmers. We will qualitatively analyze the difficulties we observe users experiencing and process descriptions they verbalize. Both will help determine the best support to mark code for extraction, what prompts are necessary to collect relevant code reuse information, and how to present the process intuitively in software to non-programmers.

Having a software solution for *finding* and *extracting* in the code adaptation process we must evaluate how it compares to other solutions. We will run a user test with four user groups: without support, a debugger with "primitive" copy and paste (e.g., no support for resolving context changes), our finding code support and primitive copy and paste, and with our full software solution. We will again measure the percentage of correct answers and average time taken to determine if our software is an effective solution.

With support for *finding* and *extracting* code we can finally approach *integration*. To support integration requires investigating difficulties users experience tailoring reused code to their wants. With another exploratory non-programmer user study, we will quantitatively and qualitatively analyze what extracted code designs and types of modifications frustrate users. Diagnosing these frustrations will enlighten any redesign of extraction to minimize these problems and design of further interactions to assist in integration.

4. Current Status

Currently we are prototyping interactions for *extracting* code "scripts" through Hastings and seamlessly using scripts in another program. Users bound the code containing features they desire and use this to create a script. Each script has roles corresponding to the actions taken by a given actor and these roles are named by the user for reuse purposes. A user can use a script with a particular program by assigning program objects to script roles and our software creates the code with the appropriate actors without syntax errors.

Additionally we are developing a debugger system for Looking Glass to be used in our Hastings evaluation user study.

5. Implications

Non-programmers in our domain are similar to non-programmers in other domains because they likely have the same preconceptions, strategies, and difficulties we identify and accommodate. Thus providing a software solution for non-programmer code adaption in one domain lays a foundation for adaptation solutions in other end-user domains (e.g., web programming, Photoshop plug-ins). These domains have examples on the web and support for adapting them will help end- users use these examples with less required knowledge.

6. References

[1] C. Canal, J. Murillo, and P. Poizat, "Software adaptation," *L'objet*, v. 12, pp. 9-31, 2006.

[2] P. Gross and C. Kelleher, "Non-programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers," To appear in *Proc. of VLHCC 2009.*

[3] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in *Proc. of CHI*, 2007, pp. 1455-1464.

[4] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proc. of CHI*, 2004, pp. 151-158.

[5] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Trans .on, Soft. Eng.*, v. 32, pp. 971-987, 2006.

[6] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M. Burnett, "Interactive, visual fault localization support for end-user programmers," *Journal of Visual Languages & Computing*, v. 16, pp. 3-40, 2005.

[7] T. Sato, B. Shizuki, and J. Tanaka, "Support for Understanding GUI Programs by Visualizing Execution Traces Synchronized with Screen Transitions," in *Proc. of ICPC 2008*, pp. 272-275.