Debugging in Envision

Research Project Description

Lukas Vogel
luvogel@student.ethz.ch

September 5, 2014

Introduction

Envision is a research project with the goal to provide a next generation integrated development environment (IDE) for object-oriented programming languages. Envision presents code in a visual environment using a combination of text and graphics. The visual environment tries to help programmers in managing large code-bases, both in terms of comprehension and navigation.

While Envision currently has functionality to edit and write code, its goal is also to improve productivity in additional tasks by integrating more tools. One essential feature of IDEs is the ability to build, run and debug programs. This project aims to improve Envision in exactly those aspects.

Debugging in file-based IDEs can be confusing as one often jumps through files and thus quickly loses the context. Code Bubbles [1, 2] is a project, which tries to improve the debugging experience. A code bubble shows a certain part of code, often a method, in a freely dragable, resizable and colored rectangle. The interface of Code Bubbles is designed to make the working set of developers explicit as an arrangement of bubbles. This is especially useful for debugging. Microsoft Research introduced a production ready Visual Studio plug-in called Debugger Canvas [3] which uses an interface based on the bubble paradigm to improve the debugging experience. Both the Code Bubbles and the Debugger Canvas authors found, in user studies, that their new approach can improve a programmer's productivity.

While both projects introduce a visual component, the bubbles, the code inside a bubble is still presented as plain text. Envision has more visual elements, it visualizes code in a mixture of text and graphics. In this project we will explore how we can create a productive debugging environment integrated in Envision. We will investigate how different visualizations and visual arrangements can improve the debugging experience.

Tasks

1 Build support

The first task is to add support for compiling Java code from within Envision and visualizing possible feedback from the compiler, such as errors and warnings.

2 Run support

As a requirement for debugging, Envision needs to be able to run a built program inside Envision. This task includes finding a visual representation of the program output. Many IDEs do this with a built-in console, which is one of several possibilities to consider. In Code Bubbles [1] there is the possibility to split console output to multiple console bubbles, which is also an interesting idea to investigate.

3 Debug back end

The next step is to debug the program while it is running. We will evaluate possible ways to connect to a Java debugger and choose a suitable one to use in Envision. We can either connect to `jdb` over standard input and output, or use the Java Debug Interface (`jdi`) via the Java Native Interface.

Then we have to implement an API in Envision, which is to be used in the front end. The API should support basic debug actions using the chosen Java interface. Basic support means support for breakpoints, reading runtime values of variables and stepping forward.

4 Debug front end

The main focus of this project is to design and experiment with an effective visual interface for debugging in Envision. We will explore ways to present stack traces, breakpoints, variable values, controls and more. This visualization should make use of the visual elements enabled by Envision.

In file-based IDEs it is difficult to show methods from different files concurrently. Envision has the possibility to freely arrange methods or other programming fragments such that they can be shown at the same time. This allows us to investigate ideas as shown in Code Bubbles, where callgraphs are created while stepping through the code in a debug session. We can experiment with the elements Envision already provides to further improve on such ideas, for example with run-time value overlays for variables and fields.

The newly introduced debugging view should provide a helpful tool to the programmer. That means the debugging view should be responsive, easy to understand, and provide a fast way to find bugs. There should be several test cases which highlight the usefulness of this debugger tool.

At a minimum we will introduce controls to start, pause, resume and stop the execution, this also requires a visual cue for the current location when we break the execution. Another core feature is the possibility to introduce breakpoints at any statement. When the execution is paused there has to be a way to read the memory values of the current context and the call trace should be shown.

Schedule

A tentative time schedule for the project

Task	Start date	Time
Compile support	15.9.2014	2 weeks
Run support	29.9.2014	2 weeks
Debug back end	13.10.2014	4 weeks
Debug front end	10.11.2014	4 weeks
Tasks completion	8.12.2014	2 weeks
Break (holidays & exam session)	19.12.2014	until next task
Write-up & finalizing	26.1.2015	3 weeks
End	13.2.2015	

References

[1] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[2] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[3] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press.