SomeTitle

Tørresen, Håvard

Supervisor:
Trætteberg, Hallvard

March 19, 2014

Abstract

Background:

Results:

Conclusion:

Acknowledgements

Contents

List of Figures

List of Listings

1 Introduction

New students may find programming in general, and object oriented programming in particular, difficult. The understanding of computers, and how they work, is not given a lot of focus in the school system, and only a few students have actually tried learning programming before coming to the university. With little or no previous knowledge, it can be hard to understand the concepts, and to get a mental model of what is going on when writing a program. Especially code not written by themselves, for example exercise frameworks, and code generated by various tools, can be hard to get a good understanding of. Traditional debuggers are not necessarily helping when detecting a runtime error, and often significant amounts of time is spent searching for the cause of a bug, instead of actually fixing it[3]. Tools that present the state of a program in a simple visual way may help to understand.

Learning goals for MMI-course (man-machine-interaction):
Introduce the student to concepts, methods and techniques for designing man-machine-interfaces, knowledge and skills in object oriented construction of graphical window-based interfaces.

Will such tools be useful in helping students to reach the learning goals of MMI and other beginner-courses? Are existing tools good enough? If not, can something be modified to better fit the purpose?

2 Task Description

Examine whether tools like JIVE can be used to aid students with their understanding of programming, and software structures. Attempt to identify changes that may be necessary e.g. default configuration, handling of visual models, performance. Which changes are actually possible to do? How to do them?

3 Prestudy

3.1 Methods

There are several ways a debugger can aid the programmer beyond just showing the current state of a program. For a fresh programmer, either in general, or at a certain project, the most useful method is probably to generate diagrams that visualize the current state, and the path of execution. I.e. some form of object-, and/or sequence–diagram. Such diagrams can make it easier to get an overview of a programs current state, and to understand how it works. In order to generate the diagrams, the tools can analyze both the source-code, and an execution trace, depending on the type of diagrams to generate.

Execution traces can also be used to enable backwards stepping of program execution. Stepping back in time allows the user to not only see the failure state of a program, but to go back and see what caused the problem, instead of adding a new breakpoint and running the program again. Each reverse step can be fairly cheap, but it may still be impractical to make large jumps in the execution history.

One can avoid the potential disadvantages of manual backstepping by using queries instead. Queries enable the user to asks the debugger about the current and earlier states of execution in a simple way. The debugger then does the work of finding what was asked for, instead of the user manually searching through the program states.

3.2 Existing Tools

There currently exists several tools that provide one or two of the methods mentioned above. GDB offers a tracing environment, but due to its command-line interface it is not necessarily easy to use on its own. The Trace Viewer Plugin for g-eclipse, uses a trace to generate visualizations of the program execution, and thus makes it easier to understand, but is designed for massive parallelism, and may not be very useful for understanding smaller programs. Whyline[2], and the Trace-Oriented Debugger[5] also utilize execution traces, but use them to enable querying, instead of providing visualizations. Additionally, Whyline exists only as a separate application, and does not integrate into any IDE. JAVAVIS[4] provides visualizations in the form of UML-diagrams, but does not provide any debugging features. Code Canvas uses an interesting way of visualizing an entire project, everything from source-code to design documents and diagrams are layered onto a large canvas, allowing easy navigation between various elements, but is restricted to Microsoft Visual Studio. Jinsight is a powerful tool built by IBM, supporting both tracing and visualization. However, it is restricted to z/OS and linux on system Z, preventing most people from using it.

JIVE seems to be the only tool that utilizes all three methods, as well as being freely available as a plugin for eclipse, making it easy to install and use. During program executiion, Jive generates a contour diagram[1], and a sequence diagram. Combined with an execution trace, it allows the user to jump back and

forth in the execution, and have the diagrams updated accordingly. Querying is supported through the JQL, and is accessed trough a simple graphical interface with templates for the most common queries, as well as a text-field allowing the user to write any kind of query.
Due to all the extra work being done when using jive to debug a program, the performance is not always acceptable. For small non-interactive programs, the added waiting time may not be a problem, but larger programs may suffer from a significantly longer execution time, and even simple interactive programs can use up to a second to respond to input on a fairly powerful computer.

Jinsight
made by IBM
two components: profiler and visualizer
only for z/OS or Linux on system z
builds a trace when application is running
client connects to profiler and visualizes the trace
modified JVM?
120 minute trace limit
very powerful

Javavis
relies on the Java Debug Interface (JDI), and the Vivaldi Kernel (a visualization library)
shows dynamic behavior of running program
object diagrams+sequence diagram, UML
smooth transitions
not a debugger

code canvas (visual studio)
unites all project-files on a infinite zoomable surface
both content and info
layers of visualization - files/folders, diagrams, tests, editors, traces ++
several layers visible at the same time
search

trace viewer plugin (g-Eclipse)
g-eclipse=grid, archived project
visualize and analyze communication of message-passing programs - communication graphs
standalone/platform independent
designed for massive parallelism - MPI and similar
debugging
events are marked by different colored nodes in the graphs.

Whyline
Interrogative debugger
why did, why did not
works on recorded executions

TOD: Trace-Oriented Debugger

omniscient debugger
queries
dynamic visualizations - high-level, graph of event density

Jive
combines all fields
contour diagram - Enhanced object diagram, showing objects and their environments: fields, values, relations, inheritance, etc.
sequence diagram - generated during execution, supports zooming and folding to cope with, and hide irrelevant information, but can still become quite large.
stepping - state-saving enables fast backward stepping, and the current state is reflected in the diagrams.
queries - enabled by state-saving. Allows filtering of irrelevant information.
can be used for debugging

4 Conclusion

Glossary

Breakpoint A source code marker telling the debugger to halt program execution at a certain point. 4

Code Canvas A visualization-tool for Microsoft visual studio, showing code, diagrams and documents on a large layered ccanvas. 4

Contour diagram An enhanced object diagram, showing objects, their variables and their relations to other objects[1][6]. 4

Execution trace A log of all changes to the state of a program throughout its execution. 4

GDB GNU debugger. A multiplatform, multilanguage CLI-debugger with tracing. http://www.sourceware.org/gdb/. 4

IDE Integrated Development Environment. A software application that provides facilities for software development such as source code editor, compiler etc. 4

JAVAVIS Tool that generates UML diagrams from running java applications[4]. 4

Jinsight An advanced debugger made by IBM, supports visualization, and powerful analysis. 4

JIVE An advanced debugging tool supporting visualisation, backward stepping, and querying. http://www.cse.buffalo.edu/jive/. 4

JQL Jive Query Language, used to formuate queries within the jive debugging environment. 5

Trace Viewer Plugin todo. 4

Trace-Oriented Debugger TODO [5]. 4

Whyline A query-based debugger that provides an easy way to find out why things are as they are[2]. 4

References

[1] B. Jayaraman and C.M. Baltus. Visualizing program execution. *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 30–37, 1996.

[2] Andrew J. Ko and Brad a. Myers. Finding causes of program output with the Java Whyline. *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, page 1569, 2009.

[3] Andrew J Ko, Brad A Myers, Senior Member, Michael J Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek , Relate , and Collect Relevant Information during Software Maintenance Tasks. 32(12):971–987, 2006.

[4] Rainer Oechsle and Thomas Schmitt. JAVAVIS : Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). pages 176–190, 2002.

[5] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *ACM SIGPLAN Notices*, 42(10):535, October 2007.

[6] James T Streib. Using Contour Diagrams and JIVE to Illustrate Object-Oriented Semantics in the Java Programming Language. pages 510–514, 2010.