SomeTitle

Tørresen, Håvard

Supervisor:
Trætteberg, Hallvard

April 7, 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 how a program works, and how its components interact with each other.

During the second year of the computer science study at NTNU, there is an increased focus on projects and more complex software. Among the mandatory courses, ... 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 at a breakpoint chosen before running. 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, see the contents of objects and how they relate to each other, and to understand how the various components work together.

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. For a general overview, a class diagram, showing how object types are composed, can be generated by simply analyzing the source code. But in order to get a more specific view, for example of how the different components interact, an analysis, either in real-time or from an execution trace, of a running program is needed. Such an analysis can be used to generate object diagrams that show how objects interact and what values they have at a given time. One can also get a sequence diagram, showing what methods were called, and in what order.

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[4] 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[7] 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[5] 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[6] 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.

4 Enhancing JIVE

In this section I will first explain the various features of JIVE, before identifying any shortcomings, and suggest improvements.

4.1 features

The most apparent feature of JIVE is its generation of contour- and sequence-diagrams during debugging of a program. The diagrams are updated according to the current state of the debugged program, so that the backtracking functionality allows you to see the entire execution graphically step-by-step.

Contour-diagrams are based on an old technique to give semantics to Algol-like languages. The basis has been extended to support modern concepts, such as object-oriented programming. Objects are represented by a box, or contour. Within the contour, the objects variables are shown, with name, type and value. The contour also uses arrows to point at other contours that are related, e.g. an other object representing the value of a variable, or an enumerator. Inheritance is shown by putting the contour of an object within the contour of the extended object. Object instances are kept separate from the contours representing inheritance, but will have relational links when necessary. Method calls are also represented in the diagram, in their own contours, linked to the calling object. JIVE offers to hide some of the information, such as inheritance or the composition of objects, in order to make the diagram smaller, and easier to read. Something that can be especially useful when working with larger programs, with many objects and relations. Visibility is aided by the use of colors

The sequence diagrams are fairly standard, with threads and objects represented by boxes on a row at the top, each with a vertical life-line. The actual sequence is shown with a thicker lifeline, with arrows representing method-calls. In order to differentiate the threads where the execution is happening, the sequences are colored with the same color as the thread- box the sequence originates from, regardless of witch objects and methods are involved. In order to cope with size, it is possible to collapse a method call, so that the lifeline of the called object is hidden.

Both of the diagrams can be saved as a high-resolution image at any time, so that one can look at earlier diagrams, instead of being forced to view them through JIVE. This also helps to visualize any changes made to a program, and to see what the effects are on the program flow. They also share the ability to zoom in and out, further helping with the handling of larger diagrams.

Closely related to the diagrams, is the ability to quickly jump backwards and forwards between execution states. This is enabled by the trace-log, containing an entry for every single event that occurs during execution of a program. Each event is assigned an identifying number, in ascending order, and information about thread, type, caller, target, and the location of the source-code is stored. The log is used as the basis for the models that make up the diagrams, and can be saved as both XML- and CSV-files for later use. As mentioned in the Prestudy, logging every event has a significant impact on run-time performance,

limiting the size of the programs that can be used in a meaningful way. On the other hand, it allows the almost instant jumping between recorded states, as opposed to techniques that save a snapshot at predefined intervals, requiring the program to be run from the snapshot-state to the desired one, even if it is just a single step backwards.

Finally, the trace-log enables the use of queries to search for specific events in the execution. The queries are presented through a new tab in the eclipse search-window, and comes with several pre-defined templates to simplify searching. For example, searching for when a certain variable gets a certain value, only requires the user to specify the variable-name, its parent, and the value. It is also possible to write custom queries, using the JIVE-Query-Language.

5 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][8]. 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[5]. 4

Jinsight An advanced debugger made by IBM, supports visualization, and powerful analysis[6]. 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 Tool to visualize and analyze communication of parallel message passing programs
citeKranzlmuller. 4

Trace-Oriented Debugger Trace-Oriented Debugger. A debugging tool that executes queries on program traces[7]. 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] Dieter Kranzlmüller and Christof Klausecker. Scalable Parallel Debugging with.

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

[6] Wim De Pauw and John Vlissides. Visualizing Object – Oriented Programs with Jinsight. pages 541–542.

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

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