Visualising Exemplary Program Values

Marcin Stefaniak
Institute of Informatics
Warsaw University
stefaniak@mimuw.edu.pl

ABSTRACT

We describe an idea of a tool to aid software developers, similar to tracing and software visualization. The tool monitors a running program and log some values of its variables. The exemplary values, chosen by the tool, are later displayed onto the source code. Each variable occurrence in the source code is visualized with a few examples of its runtime values. We are unaware of such a tool implemented already, and the question which values should be selected seems interesting.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, tracing*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*documentation*

General Terms

Languages

Keywords

Tracing, Variables, Visualization, Branches

1. INTRODUCTION

During software development, working with existing code is as important as creating new code. To understand existing source code, good documentation is very helpful. Human-created documentation and comments are prone to inconsistence with ever-changing program codebase. Some pieces of documenting information can be automatically generated by program analyzers. For example, in programming languages with type inference, like ML language family, the type information of variables is usually omitted. However, a clever IDE could infer all types by static analysis and display variable type information. It could be displayed, for example, on hovering mouse over a variable.

The Java language has no type-inference. However, Eclipse, industry-wide IDE for Java actually performs that kind of

Copyright is held by the author/owner(s).
ESEC/FSE'07, September 3–7, 2007, Cavat near Dubrovnik, Croatia.
ACM 978-1-59593-812-1/07/0009.

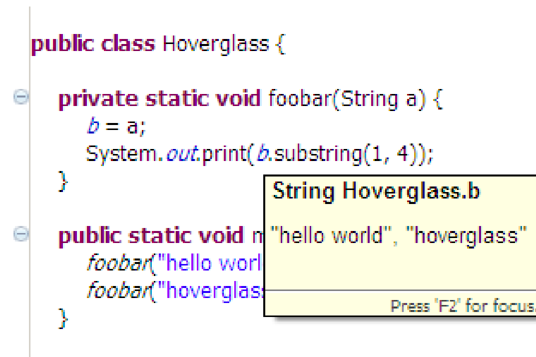

Figure 1: A hover feature in Eclipse.

visualization. The type of a variable is shown in its „hover" box, so the programmer does not have to look it up at the origin. Note that in the figure of hover box, the sample variable values „hello world" and „hoverglass" were added manually.

Similarly, runtime analysis can be used to gather information about program in order to display it to the programmer in a similar fashion, for the purpose of making the program more readable and better documented. We argue that exemplary runtime values of program variables should be shown in the „hover" mode.

Knowing the type of a variable is useful for the programmer trying to guess the meaning of program source code. In dynamic-typed languages there is no type information in the source code. A sample runtime variable value could reveal to the reader relevant information. And even with static typing, it happens frequently that the „official" type is hardly meaningful, for example when complex structured data records are represented as a string.

Moreover, seeing consecutive values of variables could help understanding functions called from the browsed code. Sometimes system or third-party libraries are scarcely documented, and sometimes it's just easier to learn little details of a function semantics (like indexing base of substring function) from comparing its arguments and result.

It is impractical to collect all values of variables, because it would generate megabytes of events per second. And in fact nobody would like to watch all of them. Instead, we would like the tool to pick just a few of them. Which of them should be chosen? - this question remains open. Which of the values would be most interesting for the programmer reading the source code?

We are presently unaware of a tool fitting the above description. Neither have we implemented it yet, so for the time being we are advocating a vaporware. Nevertheless, we need to pick up a codename for such a hypothetical tool, so let us call it neatly „hoverglass".

2. RELATED WORK

Program monitor is a program that gathers information about a running program. For example, automated program tracers, software profilers, code coverage analyzers are all program monitors. Program tracing is a specialized use of logging to record information about program execution, particularly for debugging purposes. Profilers measure the performance of a program by counting the frequency or duration of executing code fragments. Code coverage analysis answers the question which statements, conditions or paths of code were executed.

Observe that size of the result of profiling and code coverage is usually proportional to the size of the source code, while for program tracers it grows linearly with execution time. Hoverglass is a kind of a program tracer that is expected to produce result of the size roughly proportional to the size of the source code.

Program monitoring is a cross-cutting concern. It is usually implemented by code instrumentation. Code instrumentation is a transformation of (source or compiled) code which inserts at certain points snippets of code necessary for monitoring. Other approaches include virtual machine monitoring and runtime code modification.

2.1 Daikon

Daikon [4] is an implementation of dynamic detection of likely invariants, developed at MIT. It runs a program, observes the values that the program computes, and finds properties that held true during execution. The properties are mostly algebraic equalities and inequalities, and may involve one or more variables. The values are observed and stored by a front-end, and later analyzed by the Daikon back-end. There are several Daikon front-ends for observing program values in different languages, including mainstream ones (C, C++, Java). We could reuse Daikon front-end components, although not directly, for example the Java front-end lacks observing local variables.

Our approach differs from Daikon, because we don't want to store all the program values; we are looking for an on-line algorithm. Also, hoverglass results are complementary to Daikon, i.e. Daikon outputs general hypothesis about program behavior, while a hoverglass tool reports special facts. So if a variable `urlString` is assigned several different URL strings, displaying one of them to a programmer would let him recognize the variable kind to be URL, while Daikon would not detect it unless extended with a URL-specific invariant module.

2.2 Bidirectional debuggers

A classic debugger allows to step along the flow of program execution, only forward, while a bidirectional debugger allows also backwards steps. This can be achieved by logging system calls, re-execution and occasional checkpointing [2]. It can be achieved also by history logging, that is, tracing all necessary events happening during program execution. The event history can be later replayed. An example of bidirectional debugger is the Omniscient Debugger [6]. It was developed a few years ago for the Java language and its instrumentation approach could be reused.

2.3 Monitoring languages

Tracing all program events is ineffective, if its purpose is a further analysis that relies on a subset of events. In such a situation, only some of the program events should be traced and passed to the analyzer. It is possible to implement dedicated tracing code instrumentation for each program monitor. To simplify it, recently [5] proposed "tracer driver" architecture. In their approach, a general tracer filters program events based on event patterns specified by analyzers. Earlier example of language for program monitoring is UFO (a.k.a. FORMAN) [1], developed for Alamo monitor architecture.

Since we claim that a hoverglass tool could adapt itself to maintain low overhead, a similar architecture could emerge in the design of such an advanced hoverglass implementation.

2.4 Software Visualization

Software visualization is the part of computer science aimed at displaying characteristics and behavior of various aspects of computer programs. Most of the research so far has been concerned with fancy algorithm animations and data structures diagrams. Hoverglass is a kind of software visualization that is tightly bound with the source code, and can be displayed onto the source code itself.

3. USAGE EXAMPLES

There are two quite different activities that motivate for hoverglass tool: test-driven development and source repository browsing.

3.1 Test-driven development

In test-driven development, a programmer writes tests before writing the actual code. In this case, (s)he is able to test the code as it is being developed. Testing the code could be performed under surveillance of hoverglass, and when a test fails, the exemplary variable values could be viewed without re-running it, likely unveiling vital clues for the reasons of test failures. This is primarily debugging activity. It would most likely happen within an IDE, and so enhancing user-interface features would lead towards a bidirectional debugger solution.

3.2 Source repository browsing

Second case is less obvious and comes from the practice of teamwork, where the source code is shared, often in a version control system. Especially in agile methodologies of software engineering code sharing is encouraged. In this environment one may need to look up and understand the code that other people wrote, which is mentally difficult. The codebase is most likely not faulty, so it is not a debugging activity. However, it is an itch to read a code that has been already executed and tested without being able to see example variable values. And we would like to scratch this itch.

If a software development team uses unit tests and version control system, they probably run the tests automatically after code changes are checked in. This tests could be performed under surveillance of a hoverglass tool, and sample variable values stored for later code browsing. Since the

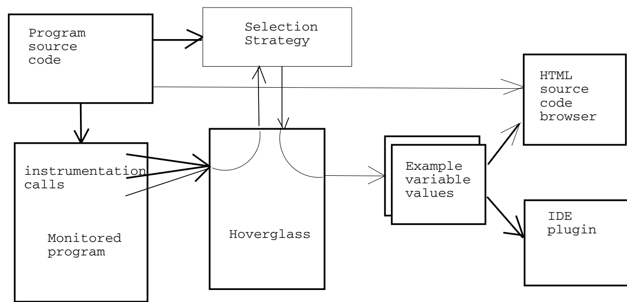

Figure 2: Hoverglass system diagram.

test suite is executed off-line, the efficiency overhead could be larger, but it should be fully automated.

Note that it is possible to browse the source repository in a web browser instead of an IDE, hence less rich user interface should be used in this context.

4. DESCRIPTION OF HOVERGLASS TOOL

4.1 Tool framework design

We strive towards proof-of-concept implementation of Hoverglass tool for Java language. Programs in Java are executed under virtual machine, so it should be fairly simple. There are already examples of successful instrumenting Java code in many projects, including Daikon and Omniscient Debugger. Code instrumentation may be performed using BCEL (byte-code engineering library) [3], which operates on Java bytecode level. The bytecode have to be compiled with debugging information enabled, so that variable names and number of lines are available.

Hoverglass tool core consists of three separated parts: instrumenting the code, runtime values selecting, and visualization of exemplary values.

The selection module is called by the instrumented code and decide which of them should be remembered for later presentation. An interface for selection algorithm is designed, so as to separate selection strategies from the low-level internal details. The exemplary values are stored for visualization using an easy data format. Variable occurrences could be more or less identified by their names and source line numbers. There can be different visualization modules, e.g. HTML generator or IDE plug-in. The hoverglass framework is defined by selection algorithm API, and visualization data format.

4.2 Execution time overhead

When observing all variable values, the program execution may be slowed considerably. The Omniscient Debugger [6] reports execution overheads from 2x to as much as 300x. The factor depends greatly on the type of code being instrumented – how often it calls other (non-instrumented) code, and whether the computations fits inside the memory cache. We shall call the code with large overhead „dense” and the kind of code that has little overhead „sparse”. Usually the „kernel”, „low-level” code is dense, while application, scripting, high-level is sparse.

Of course, too large overhead is rarely acceptable. Usually it is addressed by the fact that only part of the code is interesting enough to be instrumented. For example, system libraries and certain trusted, dense parts of code are exempted from the analysis. In an industrial-quality tool, specifying the parts of the program to be instrumented is a „must have” feature. This is an issue with all dynamic analysis tools, not only this one. However, we can go further than that.

Unlike other runtime analyzes, the program does not have to be monitored constantly. The expected result of hoverglassing is gathering at least one example value for each variable occurrence, so at the very least it could log only the first value and then disable itself. A smart hoverglass tool could profile its execution and disable tracing for too dense code fragments. This way a low overhead could be kept without sacrificing quality hoverglassing for non-dense code.

This technique could be realized by having each fragment of code compiled in two versions: instrumented and uninstrumented. When too much time is spent within the instrumented version, execution flow would switch to the latter. Self-modification of code could make it almost as performant as the original code. For classic virtual method call, it is as simple as changing an entry in the virtual method table. Under virtual machine, it could be achieved by advanced tweaking of just-in-time (JIT) compiler.

The profiling argument is useful also for the other end of the code spectrum. If the tool could detect sparse and seldom visited program fragments, it could apply more sophisticated and computationally expensive analysis. Hence, it is justified to research a wide range of variable value selection strategies - simple and fast as well as complicated and ones.

5. EXEMPLARY VALUES SELECTION

It is not obvious which values should be picked by a hoverglass tool, and various strategies for this could be proposed. Some of them could be randomized, and some of them deterministic. Notice that for deterministic strategies users could gain some information from the fact that certain values are *not* displayed.

Another issue is whether the variable values are chosen independently of each other, or not. Simple strategies treat each variable separately, while more advanced ones could relate them, for example gathering all from the same execution time. A trivial strategy would store k randomly chosen values for each variable occurrence.

5.1 Picking representative values

Let us enhance the trivial strategy, staying with independent selections. Since we want to pick values that are exemplary for a variable occurrence, they should be different. Moreover, for certain distinctiveness features of variable values, we would like to log at least one representant for each distinct feature value. The collection of such features includes, but is not limited to:

- the runtime type of variable value
- is variable value null?
- for string variable, is it empty or not?
- for numeric variable, is it negative, zero, or positive?
- for small enumerated types, the value itself is a feature

For simple type variables, the number of interesting features is small enough to have all of them considered. On the other hand, when number of possible feature values exceeds the limit of space for exemplary values, as it is possible with complex types, some of them have to be dropped.

5.2 Conditional branch-based selection

Conditional branch instructions evaluate an expression in order to divert the flow of program, so their conditional expressions are very important to program behavior. So, if a variable is contained in branch instruction expression , the result of expression should be treated as a feature for this variable.

For example, if for the following piece code

```
function foo(x)
  if (x > 5) {
    print x
  }
}
```

the function `foo()` is called with values $\{2, 3, 7, 8\}$, one value should be selected from the set $\{2, 3\}$ and one from $\{7, 8\}$, so that values greater then 5, and those that are not, are represented. This example is an easy one, because the branching expression depends only on one variable. When the expression involves other variables or function calls, it is not clear that a particular value contributed much to the result.

Of course, this approach is not restricted to the `if-then` instruction. It applies to other branching instructions, like conditional loops, as well. The branching expression of `switch` statement, is not boolean, but rather multi-valued; naturally, each its `case` branch should be treated as a different value of distinctiveness feature. Also, exceptions thrown during evaluation of expression with variable values could be treated similarly, because they create an (implicit) opportunity of branching.

5.3 Entangled variable occurrences

Strategies that select values independently for each variable occurrence is not enough. When an exemplary value of a variable is chosen, it is useful also to preserve the values of neighbor variable occurrences at that moment. This could provide useful feedback information about the flow of data. For this, we could call such variable occurrences „entangled".

It should be noted that such entangling of variable occurrences depends solely on the monitored program structure. Therefore, its influence can be pre-computed at compilation (instrumentation) time. However, this would complicate hoverglass framework implementation considerably, and is likely to be omitted from the first implementation release.

6. FURTHER WORK

Certainly, a major goal is a proof-of-concept implementation of the hoverglass tool framework. Enhancing it with self-profiling in order to maintain low overhead seems a challenge in both design and development.

As far as the strategies of variable selection are concerned, it seems a tip of iceberg. Apart from conditional branches, it is possible that other program structures, like loops, could influence the way exemplary values are visualized. Also, there could be a need to gather exemplary values of subexpressions, not only variable occurrences. It is not strictly necessary because a subexpression can be always extracted into a local variable by the developer, but it could be useful for certain kind of expressions, for example accessing a field of a structure. And apart from variables of simple type, handling and visualizing complex structures, objects and arrays should not be underestimated.

The visualization problem contains a flavor of „human-computer interaction" field. So this approach should be evaluated and driven by feedback from human users, namely software developers. That explains why the implementation should be developed within a widespread environment (like Java).

7. REFERENCES

[1] M. Auguston, C. Jeffery, and S. Underwood. A monitoring language for run time and post-mortem behavior analysis and visualization. *CoRR*, cs.SE/0310025, 2003.

[2] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM Press.

[3] M. Dahm. Byte code engineering with the bcel api, 2001.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[5] L. Langevine and M. Ducassé. A tracer driver for hybrid execution analyses. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 143–148, New York, NY, USA, 2005. ACM Press.

[6] B. Lewis. Debugging backwards in time. *Proc. of the Fifth Int. Workshop on Automated Debugging*, 2003.