A High-level View of Java Applications

Eric Bodden
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF, UK
eric@bodden.de

ABSTRACT

Static analysis of object-oriented applications has become widespread over the last decade, mainly in the context of compile-time optimizations. The paper describes how static analysis of virtual method calls can be employed to provide a high-level view of Java applications. The result is a method call graph that can be built from either source or bytecode, and a graphical browser that enables the user to analyze control flow and the coupling between classes and packages in an intuitive fashion, thereby supporting application design as well as refactoring and debugging. In order to achieve the necessary bijection between source and bytecode representations of classes, we implement a new approach based on source code pre-processing.

Categories and Subject Descriptors

D.3.2 [**PROGRAMMING LANGUAGES**]: Language Classifications—*Object-oriented languages*

General Terms

Design, Languages, Human Factors

Keywords

Static analysis, call graph visualization, Java

1. THE GOAL

Object-oriented programming, combining inheritance and polymorphism, provides a powerful means of describing the world. Although object-oriented programming can produce reusable, modular, well structured code, program development and maintenance is still hard, especially for systems that consist of many objects interacting in complex ways. Our goal is to provide programmers with a tool to allow exploration, at a high level, of the relationships between objects used in their program. Such a view should reveal the caller/callee relationships between all methods[1] in a Java application. Once this relationship is established, one can examine how methods interact. The view should be intuitive, and should allow the programmer to explore these relationships in order to understand the consequences of a change to one class on this and other classes.

The mathematical entity for such a relation is a directed graph, here called a *method call graph*; it may be cyclic in the case of direct or indirect recursion. The graph comprises nodes representing methods, and edges representing method calls. In order to build such a call graph, we need to determine method call sites and those methods that may be called at each such site.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

```
class Library {
  void newItem(boolean isBook, String title) {
    LibItem item;
    if (isBook) item = new Book();
    else        item = new DVD();
    item.setTitle(title);
    ...
  }
}
```

Figure 1: The need for virtual method calls.

```
void aMethod() {
  LibItem item = new Book();
  LibItem otherItem = new DVD();
  otherItem.setTitle(...);
}
```

Figure 2: More precise type analysis is possible.

2. DYNAMIC DISPATCH

Like other object-oriented programming languages, Java has one awkward property: *virtual method calls.* Java must determine, for each method call, the run-time type of the object to which this method belongs. In Java this is implemented by the `invokevirtual` bytecode. For example, the `newItem` method shown in Figure 1 creates new objects whose type depends on the parameters passed. The method `Book.setTitle` is called if and only if `isBook` is `true`, otherwise `DVD.setTitle` is called. Virtual method calls are not just useful but necessary. In this case, our

[1]The notion of *method* includes constructors.

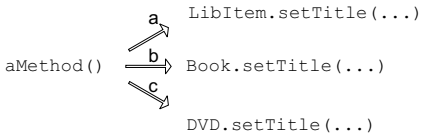

Figure 3: Virtual calls derived from the example in Figure 2.

tool must allow the programmer to explore (at least) the edges from `Library.newItem` to both `Book.setTitle` and `DVD.setTitle`.

3. CALL GRAPH CONSTRUCTION

Three well-known techniques for call graph construction have been suggested in the literature. The simplest of these is Class Hierarchy Analysis (CHA) [2]. CHA analyses a program statically to construct a graph of all possible edges between callees and callers. For Figure 1 it gives edges from `newItem(...)` to the `setTitle(...)` methods in the class `LibItem` and *all* its subclasses, since `item` could be of any subtype at run-time. In our case, this would create all three edges of Figure 3, which is rather wrong and inefficient.

Fortunately, such graphs may be optimised if the run-time type of a caller can be determined at compile-time, as in the example from Figure 2. Although the classes `Book` and `DVD` are subclasses of `LibItem`, for the method call `setTitle(...)` it is clear at compile-time that the `otherItem` is a `DVD`.

Rapid Type Analysis (RTA) [1] is somewhat slower in practice than CHA but can eliminate some edges from the call graph. RTA first builds a call graph using CHA and then deletes any edges to classes for which no objects are instantiated inside the calling method. This is still quite quick and often trims the invoke graph substantially but usually still leaves 'dead edges'. In our example, only the edge `a` would be deleted.

Finally, Variable Type Analysis (VTA) [4] is the most expensive but also most sophisticated of the three algorithms. VTA uses type propagation to determine the static type. From every instantiation (by `new`), the type is propagated to the method call by examining all statements that could change the type (such as assignments). In our example, this would delete all edges but `c` in Figure 3. The drawback of VTA is that it is expensive of time and space.

4. IMPLEMENTATION

We considered analysing both source and bytecode. Bytecode leads to a more accurate representation of run-time behaviour (consider compile-time optimization), is easier to parse, quicker to process and always accessible. Any additional information provided by source code is unnecessary for our analysis. The chief drawback of bytecode is that it must be converted into a human readable form for display.

Our work combines the analysis techniques mentioned above with a powerful visualization tool that supports refactoring by enabling the user to investigate the application in a user-friendly way. Our analysis framework was provided by the SOOT bytecode analysis package [5] from the Sable[2] group at McGill University, which provides all three analyses. Call graph visualization is provided through OpenJGraph[3].

Our tool JAnalyzer[4] enables developers to load and store projects containing source or bytecode files. Source files are pre-processed into an intermediate representation (one expression per line) in order to gain a bijection between single expressions in the source code and lines in the bytecode. This is performed using a JavaCC generated parser whose output is used directly to highlight syntax errors. As a result, the user can invoke the bytecode analysis by clicking on the appropriate sourcecode item. The size of the call graph can be limited by selecting specific methods as analysis entry points and filtering 'uninteresting' classes; filtered classes are considered as *phantom* classes in SOOT and represented as leaves of the call graph.

Once the graph is build, the user can enter the high-level view by simply clicking on a method invocation in the source code. A subset of the call graph, centred on this invocation, is displayed and can be further explored. Remaining in this high-level view, the user can follow control flow by following edges of the graph displayed and can investigate further method calls by direct interaction with its method nodes.

5. CONCLUSIONS

We believe that we have provided a useful tool to support program debugging and refactoring [3]. Once the call graph has been constructed, it is sufficiently fast to allow direct exploration of the relationships between methods. Complexity is controlled, and comprehensibility enhanced, through filtering out uninteresting classes and by restricting the graph displayed to the nodes currently of interest. We intend to integrate it into the Eclipse IDE[5] and to provide enhancements such as a browser-like history for the visual graph, and incorporation of new SOOT 2.0 functionality.

6. ACKNOWLEDGEMENTS

Much gratitude goes to Richard Jones who gave enormous contribution to the research of this project and helped in many ways to find appropriate solutions for many problems the team came across. Also many thanks to my colleagues Piotr Piasecki and Jian Yang who contributed to JAnalyzer's research and development.

7. REFERENCES

[1] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA '96*, pages 324–341, November 1996. ACM Press.

[2] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP '95, Lecture Notes in Computer Science*, volume 952, pages 77–101, 1995.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *OOPSLA '00*, pages 264–280, 2000. ACM Press.

[5] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. SOOT - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.

[2] http://www.sable.mcgill.ca/

[3] http://openjgraph.sf.net

[4] http://janalyzer.bodden.de

[5] http://www.eclipse.org