# JAnalyzer

# A Visual Static Analyzer for Java™

**Eric Bodden**

eric@bodden.de
University of Kent, Canterbury, United Kingdom
Proposed by Richard E. Jones and Andy M. King
{R.E.Jones,A.M.King}@kent.ac.uk

As contribution for the SET Awards 2003, category computing.

Eric Bodden is a student of the University of Technology Aachen (RWTH), Germany. While spending a year abroad, he was Project Manager, Software Architect and Lead Programmer of a project group consisting of three third-year students at the Univesity of Kent, where he obtained a Diploma in Computer Science (Distinction). He is currently working for the IBM Java Technology Centre in Hursley, UK, and will continue his studies in Germany in October.

The contributions to this project by his two fellow students, Piotr Piasecki and Jian Yang, are clarified in this report.

## 1. Introduction

Object-oriented (OO) programming has well-known benefits, producing reusable, modular, well-structured code. Nevertheless, program development is still hard, especially for large programs that consist of thousands of interacting objects. Faults in one method can propagate to others defined in different classes or even different packages. The programmer would benefit from a high-level, intuitive, graphical view of these method dependencies. Such a view would aid refactoring [1] by revealing the degree of coupling between different parts of the program as well as save debugging time by allowing design faults to be visualized at implementation time in a straightforward way. Our research concentrated on Java due to its increasing popularity and platform independence.

In contrast to classic imperative programming paradigms, the development of such a view is a non-trivial task for OO languages, because methods are typically invoked through *dynamic dispatch* — the type of the object on which the method will actually be invoked is not known at compile time. Such polymorphism has the benefit to the programmer of reusable code, but means that the relationship between caller and callee is *1:many* rather than *1:1*. A type analysis for Java is therefore required to synthesise a *set* of possible types for each object identifier in the program. Inferring these sets is a complex task, which we explain further below.

Our research concentrated on Java due to its platform independence and increasing popularity. Our standalone tool, JAnalyzer, aids program development by:

- construction of call-graphs by state of the art analyses
- visual representation of inter-dependencies between methods, thus aiding refactoring
- comprehensible and responsive views of even very large call-graphs

## 2. The goal of the project

As a final-year undergraduate project at the University of Kent, a team of three students developed a tool intended to enable software architects and programmers to gain a high level view of their application through static analysis of their Java code. This analysis should include a representation of the class hierarchy as well as the caller / callee relationship between methods; these relationships were to be displayed visually as a graph.

## 3. Static call graph analysis

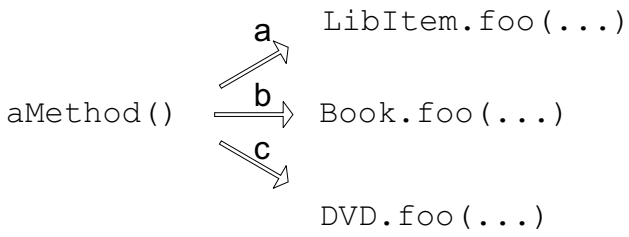
Research led to three approaches to type analysis: Class Hierarchy Analysis (CHA) [2], Rapid Type Analysis (RTA) [3] and Variable Type Analysis (VTA) [4]. CHA constructs a call graph through application of a type analysis, based simply on the class hierarchy of the program. However, its analysis is overly conservative, leading to a call graph that is larger than necessary, since the set of classes that CHA associates with an object identifier may include cases not realised (i.e. classes of which no object has been created).

RTA and VTA prune the call graph in order to eliminate such *false positives*. By tracking object instantiation (RTA) or control flow (VTA), these techniques are able to prune the call graph substantially.

```
public void aMethod(){
  LibItem item;
```

```
    item = new Book();
    item = new DVD();
    item.foo("whatever");
  }
```

In this example, `Book` and `DVD` are considered to be subclasses of `LibItem`: both override the method `foo`. CHA would construct the graph for this method as:




As one can see, neither `LibItem.foo()` nor `Book.foo()` is ever invoked. RTA will remove edge *a*, since `LibItem` is never instantiated. VTA will also remove edge *b* by propagating the type of `item` to the call `foo`.

## 4. Analysis of tools and libraries

An early decision was whether to base the tool on source code or byte code representation of classes. Bytecode leads to a more accurate representation of run-time behaviour (taking compile time optimisations into account), is easier to parse, quicker to process and — most important — always accessible. Any additional information provided in the source code is unnecessary for this analysis. The chief drawback of bytecode is that code must be converted to a human-readable form for display by the tool.

Research into call-graph construction tools led to the adoption of the bytecode optimization package SOOT [5], which provides CHA and VTA. Although these components represent the state of the art, they may be enhanced or replaced as research proceeds. Once the SOOT package has constructed the call-graph, it can be queried in milliseconds, promising excellent performance for its visual representation. An added benefit is the SOOT notion of *phantom classes* which replace sub-graphs by single leaves in the call-graph, thereby leading to sparser graphs and faster analysis and display.

## 5. Implementation details

The project was designed in UML, using the CASE tool Together™ [1]. The application has three major components:

- File management, pre-processing and compilation of source files
- Bytecode analysis, call graph construction, pruning and query
- Graphical user interface and call graph visualization

[1] www.togethersoft.com.

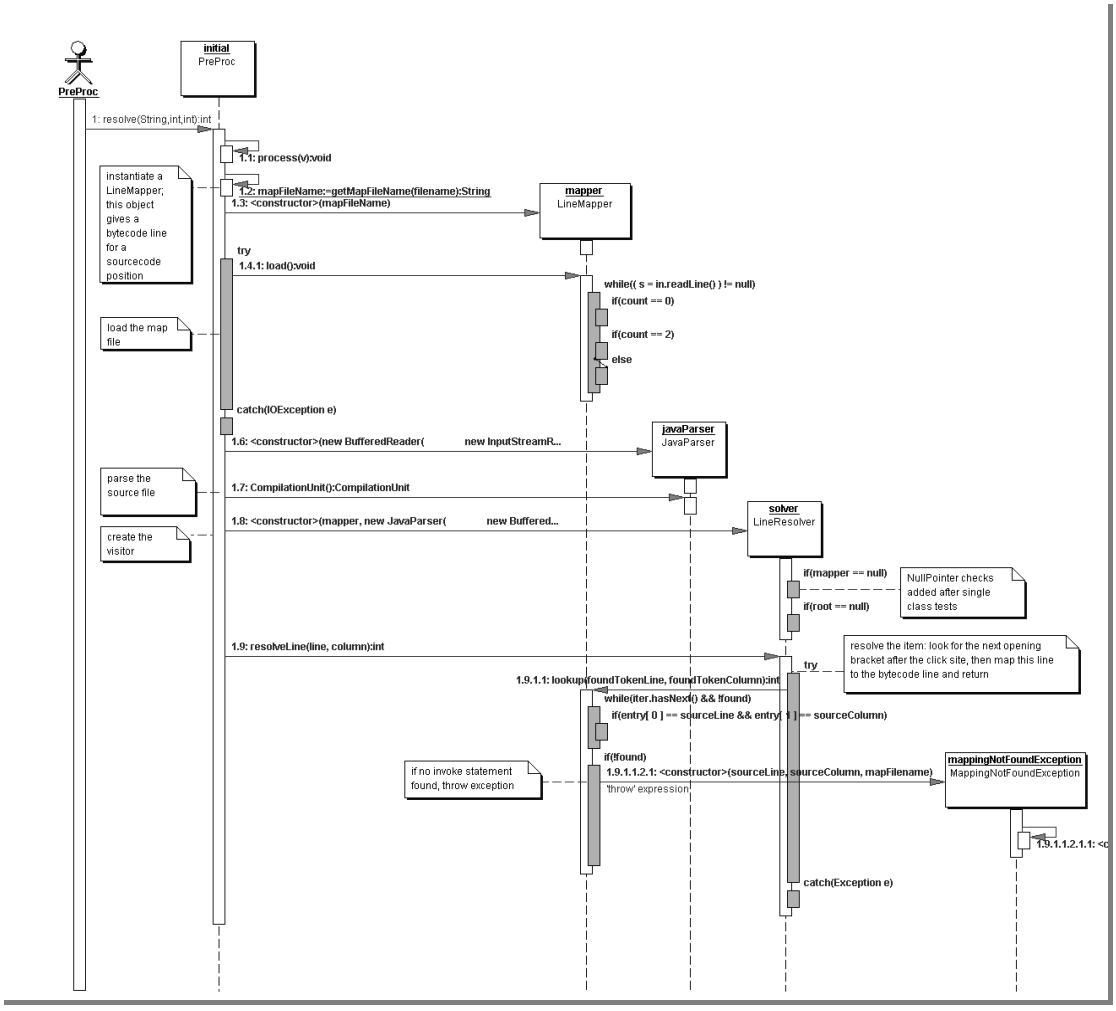



Figure 1: Sequence diagram illustrating how a bytecode line is to be resolved after a mouse-click.

## 5.1. Pre-processing source code and bytecode

Analysis of Java bytecode is surprisingly awkward because bytecodes identify only the *line number* of the corresponding statement in the source code. Unfortunately, one line of source code may map to several bytecodes, especially if that line contains more than one expression. However, a key requirement of the project was to enable the user to switch to a high-level view of the call graph just by clicking on source code identifiers.

Different approaches were considered, such as propagating types to the source code level. Our solution was to build a source code parser with JavaCC[2] to transform the source code prior to compilation in such a way that each line contains exactly one expression. The original position of any expression in the source code was logged in a hash-table, which was also used for reverse lookup. Using this technique, the static type of an object can be determined by finding a match of the user's click position (line/column in the source code) in this hash-table, thereby identifying the corresponding bytecode line (see Figure 1). This bijection between source code and bytecode proved to be reliable and highly efficient, as pre-processing and compilation are performed just once for each source code project.

[2] JavaCC is a LL(k) parser generator.

The program is analysed in several stages. In the first stage, the user to specifies a project containing either source code or bytecode files (which may be contained in a JAR archive).

In the second stage, the source code (if given) is pre-processed and compiled according to classpath and compilation parameters previously set by the user. The pre-processing stage includes syntax validation: any syntax error is highlighted in the GUI. Otherwise, the pre-processed and compiled classes are added to SOOT's internal representation, or *scene*. The user can now display classes in the GUI and investigate their members.

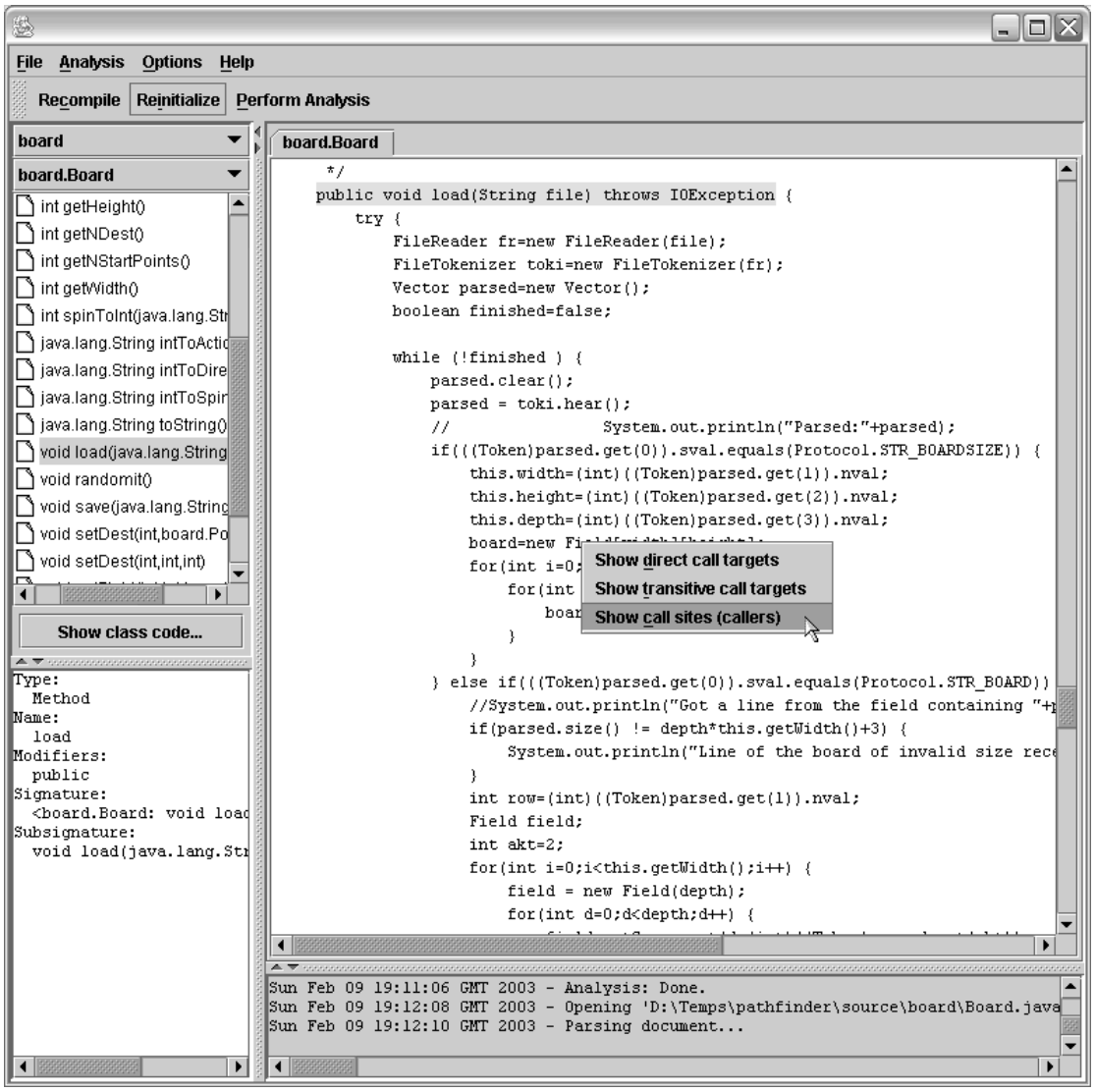



Figure 2 - Clicking on a method call in source code invokes a call-graph query.

## 5.2. Call graph production and query

In stage three, the analysis performs CHA, and optionally VTA, on the scene to produce a graph object available for browsing. All the steps described up to this point need just to be performed once.

The user is now able to open the source code of their project, click on any method calls and chose from options including show direct or transitive call targets (see Figure 2). The set of static types of the possible call targets are retrieved in order to perform the appropriate query on the call graph using internal SOOT mechanisms.

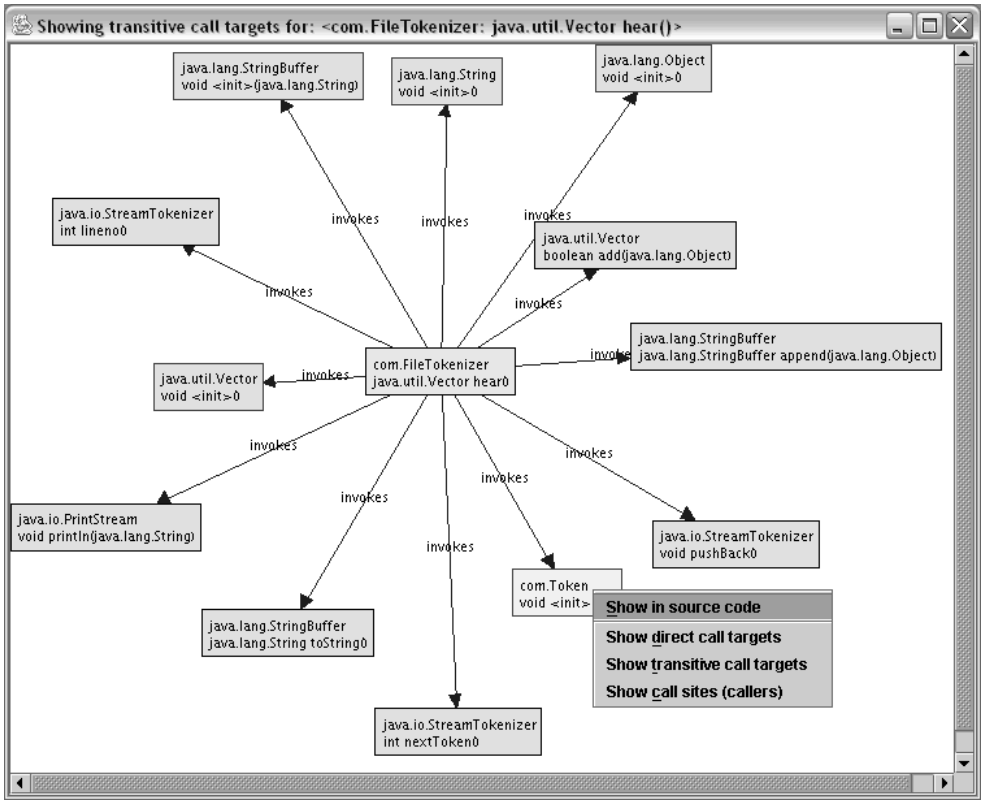



Figure 3: The call–graph generated can be investigated further through direct user interaction.

## 5.3. Rendering the high-level view

The result of the query is a representation of a subset of the call-graph, containing the methods of interest. The renderer, OpenJGraph [3], displays the graph, attaching internal objects representing methods to the appropriate nodes in the graphical display. The end user can remain in this high-level view, browsing the call graph directly through operations on nodes of the displayed graph (see Figure 3). No further retrieval of static types is necessary at this stage since the set of fully qualified calls is attached to each node in the graph already. This optimization provides crisp user interaction.

## 5.4. User guidance

It was considered to be useful to provide some guidance through the application, due to the complex nature of the topic. This was provided by highlighting those buttons in the user interface that might be suitable to be clicked in order to proceed to the next stage (e.g. *Compile* then *Perform Analysis*, and so on).

# *6. Organization of the team*

## 6.1. Project Plan

Throughout the project, the team attempted to adhere to the best software engineering practice. Structured development process and tooling were important factors for the success of the project. A strict development process was established, incorporating the following tasks:

- Requirements analysis and specification

[3] openjgraph.sf.net.

- Discovery of required and existing resources
- Research into exiting algorithms and tools for static analysis
- The choice of working with either bytecode or source code
- Design using UML class and sequence diagrams
- Breakdown and allocation of coding work
- Implementation
- Unit and Integration tests
- Documentation

Appropriate tasks were distributed between the team members according to their skills: Bodden (Project Manager) was also responsible for the software architecture as Lead Programmer, Piasecki focused on syntax highlighting, graph display, layout and testing, and Yang assumed responsibility for GUI development.

## 6.2. Quality assurance

Project requirements were specified in advance in an acceptance tests document. Standards for coding, version control, testing and test-purpose use cases were integrated into a quality assurance plan. In addition, a requirements specification for the GUI was drawn up.

A source code skeleton was generated from a UML specification (e.g. see Figure 1), based on design-purpose use cases, which again comply with the acceptance tests.

CVS was used for version control. Code was checked for compliance with the requirements specifications at reasonable intervals. After completion of a software module, unit tests were generated and applied to assure consistency of individual classes with their specification. On failure, a defect report was raised and given to the appropriate programmer; on success, an approval form was filed with the appropriate CVS version number.

The GUI specification was used to build integration tests in order to test the overall functionality.

## 6.3. Progress tracking and process integration

Progress was tracked using a project plan comprising Gantt charts and dependency diagrams that were updated on a regular basis. Milestones were defined and scheduled for fixed dates. This enabled the team to identify risks and spot problems in the development process as soon as possible.

# *7. Conclusion*

All requirements of the project were fulfilled. The final application is efficient, can use source or byte code, and is intuitive to use. State of the art type analyses allow the user to browse the call graph, either from source text directly or by following links in a graph of method calls. The tool offers user guidance, and allows filtering of 'uninteresting' classes and packages. The implementation uses creative solutions, such as the provision of a link between source and byte code through pre-processing — this approach is both feasible and efficient. The project was a success from a research point of view, and adherence to a software engineering discipline contributed strongly to its timely and complete delivery.

# *8. Outlook*

The project team is considering improvements and enhancements:

- Integration into the Eclipse[4] integrated development environment.
- Linking of the analysis capabilities to refactoring modules of Eclipse in order to provide semi-automated refactoring and better debugging.

Finally, JAnalyzer is available under GPL license and contributions to its further development are encouraged[5].

## 9. Acknowledgements

Acknowledgements go to the other two project team members who contributed to the success of this project. Much gratitude goes also to our supervisor, Richard Jones, who provided an enormous contribution to the research of this project and helped in many ways to find appropriate solutions to various problems the team came across.

The whole team is also grateful for the support from the Sable research group at McGill University as well as to Andy C. King, of the University of Kent, who provided practical advice on the use of SOOT.

## 10. References

[1] J. Brant, W. Opdyke, D. Roberts, M. Fowler, K. Beck. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

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

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

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

[5] V. Sundaresan, P. Lam, E. Gagnon, R. Vall´ee-Rai, L. Hendren and P. Co. SOOT — a Java Optimization Framework. In Proceedings of CASCON 1999, pages 125–135, 1999.

[4] www.eclipse.org.
[5] janalyzer.bodden.de.