Vizualizer:

A Code Visualization Tool for Java

Developed By:

Ryan Wiles

Erik Shreve

Presented for CS6301.502 Final Project

May 3, 2016

Contents

[Introduction 4](#_Toc450048626)

[Purpose 4](#_Toc450048627)

[Project Goals 4](#_Toc450048628)

[Technical Tool Details 4](#_Toc450048629)

[User Interface 5](#_Toc450048630)

[Graphing Library 5](#_Toc450048631)

[Java Parser 5](#_Toc450048632)

[Java Analyzer 5](#_Toc450048633)

[Text Retrieval 6](#_Toc450048634)

[Vizualizer GUI Description 7](#_Toc450048635)

[Source Selection 8](#_Toc450048636)

[Package Selection 8](#_Toc450048637)

[Class Selection 8](#_Toc450048638)

[Filter Specification 8](#_Toc450048639)

[Render Button 12](#_Toc450048640)

[Save as Image and Save as Graphviz Buttons 12](#_Toc450048641)

[Diagram Window 12](#_Toc450048642)

[Results 13](#_Toc450048643)

[Performance 13](#_Toc450048644)

[Graphs 15](#_Toc450048645)

[Future Work 16](#_Toc450048646)

[Improvement of the Graphing Engine 16](#_Toc450048647)

[Include a Metrics Sidebar 16](#_Toc450048648)

[Search Improvements 16](#_Toc450048649)

[Database Integration 16](#_Toc450048650)

[Watch Functionality 16](#_Toc450048651)

[UI Improvements 16](#_Toc450048652)

[Conclusion 17](#_Toc450048653)

[Appendix A: Survey of Other Tools 18](#_Toc450048654)

**Figures**

[Figure 1: Vizualizer GUI 7](#_Toc450048655)

[Figure 2: CAD Showing Use of Advanced Search 8](#_Toc450048656)

[Figure 3: Natural Layout (Color is Part of Reference Chains Feature) 9](#_Toc450048657)

[Figure 4: Top Down Layout (Color Is Part of Reference Chains Feature) 10](#_Toc450048658)

[Figure 5: Class Shown with Fields 11](#_Toc450048659)

[Figure 6: Class Shown with Methods 12](#_Toc450048660)

[Figure 7: JabRef CAD for "Search" 15](#_Toc450048661)

[Figure 8: FreeMind CAD for "Menu" 16](#_Toc450048662)

**Tables**

[Table 1: Code Bases Used For Performance Results 13](#_Toc450048663)

[Table 2: Code Base Metrics 13](#_Toc450048664)

[Table 3: Details of Systems Compared for Performance 13](#_Toc450048665)

[Table 4: Parsing Performance Results 14](#_Toc450048666)

[Table 5: Graphing Performance Results 15](#_Toc450048667)

[Table 6: Tool Survey 20](#_Toc450048668)

[Table 7: Tool Survey Link 21](#_Toc450048669)

# Introduction

Large software projects contain an immense amount of complexity. The comprehension of the structures and relationships contained in these systems is arguably one of the most important tasks for software maintainers. Since many humans are better able to understand the relationships found in software systems when presented to them visually, we developed a code visualization tool (called Vizualizer) to aid software developers, maintainers, and architects in being able to view these systems as graphical representations. Our focus was on providing useful graphical abstractions of the structures contained within the code under analysis while requiring a minimal amount of tool set-up.

# Purpose

The purpose of this report is to present, in detail, all work done to create the Vizualizer tool in addition to presenting results. For the tool, results include performance comparisons with other tools as well as a presentation of graphs obtained from our tool (to illustrate how the tool can be used by developers to gain understanding of a code base).

# Project Goals

Our initial goals were to provide support for accurate package dependency diagrams, CAD (Class Association Diagrams), and identification of unreferenced classes within the tool. In terms of the scalability, our target was to model med-large projects up to 500k SLOC (Source Lines of Code) written in Java 8 using an AST (Abstract Syntax Tree) Visitor. These initial goals were successfully achieved within the constraints of our approach. Larger projects can be scanned by the tool. However, graph rendering times limit usability if they involve a large number of nodes and edges.

Since we achieved our base goals, we pursued stretch goals in our project that build on the foundational work that was done. These stretch goals include the following:

1. Finding and modeling upstream and downstream reference chains to a specified depth from user selected elements for both CAD and package dependency diagrams.
2. Implement indexing and searching capabilities using Text Retrieval techniques to identify relevant elements within the scanned code to be displayed.

We accomplished both of these stretch goals.

# Technical Tool Details

In order to focus our development effort on the core goal of providing useful and accurate visualization in Java 8, we leveraged existing projects wherever possible. To this effect, we used existing open source tools and leveraged our previous work wherever possible (such as the Java code text parsing from earlier class assignments). Each major area of the tool’s technology is now presented in more detail.

## User Interface

For the user interface, a custom web page is served using the Project Grizzly webserver (<https://grizzly.java.net/>).

This choice was made in order to minimize the components the user needs to install to use the tool and to provide a cross-platform interface. The web server presents an HTML5 web interface to the client browser using the JQuery JavaScript library to make the interface interactive. The JavaScript communicates with the web server using a REST interface. The REST interface then queries the intermediate model from the Java Analyzer, discussed below, and provides responses to the client browser.

## Graphing Library

The limiting usability factor based on work done so far is in the graph rendering process regarding graphs with a large number of nodes and edges. Due to the size and complexity of the generated diagrams, the ability to perform pan, zoom, and filtering of the displayed elements is necessary for the tool to be useful in analyzing complex systems. Support for these operations was built into the tool.

Generating our own visualization layer was beyond the scope of this project; instead we included the Graphviz project, an open source graph visualization software package, for determining the layout of the nodes and edges and generating the rendered output. Specifically, we used the mdaines/viz.js JavaScript library that is a JS recompilation (Emscripten) of the Graphviz utility (see <https://github.com/mdaines/viz.js/>).

## Java Parser

For the parser we used the Java Parser project, which is a non-compiling Java parser. The key benefit of using a non-compiling parser being that the parser only looks at a single file in isolation to generate the AST and doesn’t require the user to configure classpaths or source directories.

## Java Analyzer

The Java Analyzer module performs the analysis of the AST generated by the Java Parser through the use of an AST Visitor and converts it into an intermediate model that reflects the structure of the code under analysis. The intermediate model consists of the following components:

* *Project:* This is the top-level class in the hierarchy and provides the entry point into the system. It allows files and directories to be added to the model, model validation to be started, and retrieval of different views of the model.
* *Package:* Corresponds to the package concept in Java and provides a collection of classes.
* *DependentBase:* An abstraction of common elements and behaviors used across Classes, Methods, and Blocks. Unresolved types are tracked for annotations, classes, interfaces, methods, variable names to type name. During validation, these unresolved types are resolved using optimistic type resolution and tracked.
* *Class:* Extends DependentBase and corresponds to each class, annotation, enum, and interface in the Java code. This model object tracks package dependencies, any super class, implemented interfaces, defined methods, and what other classes reference it.
* *Block:* Extends DependentBase and provides an aggregation of other blocks to support nested code blocks. It also contains a map of parameter names and types which is used for the Method model object and to implement Lambda expressions.
* *Method:* Extends Block and provides the model object for mapping methods definitions. The key difference here being support for a name and a return type.

The Java Analyzer module in addition to storing the intermediate model is also responsible for analyzing the model and generating the Graphviz formatted text for the Graphing Library. During the initial scanning of the classes the text of each class is sent to Lucene as described below.

## Text Retrieval

Lucene was used to index the source code text. The text of each class is associated with its respective class in a single Lucene index. Prior to adding to the index, the text is scrubbed as follows:

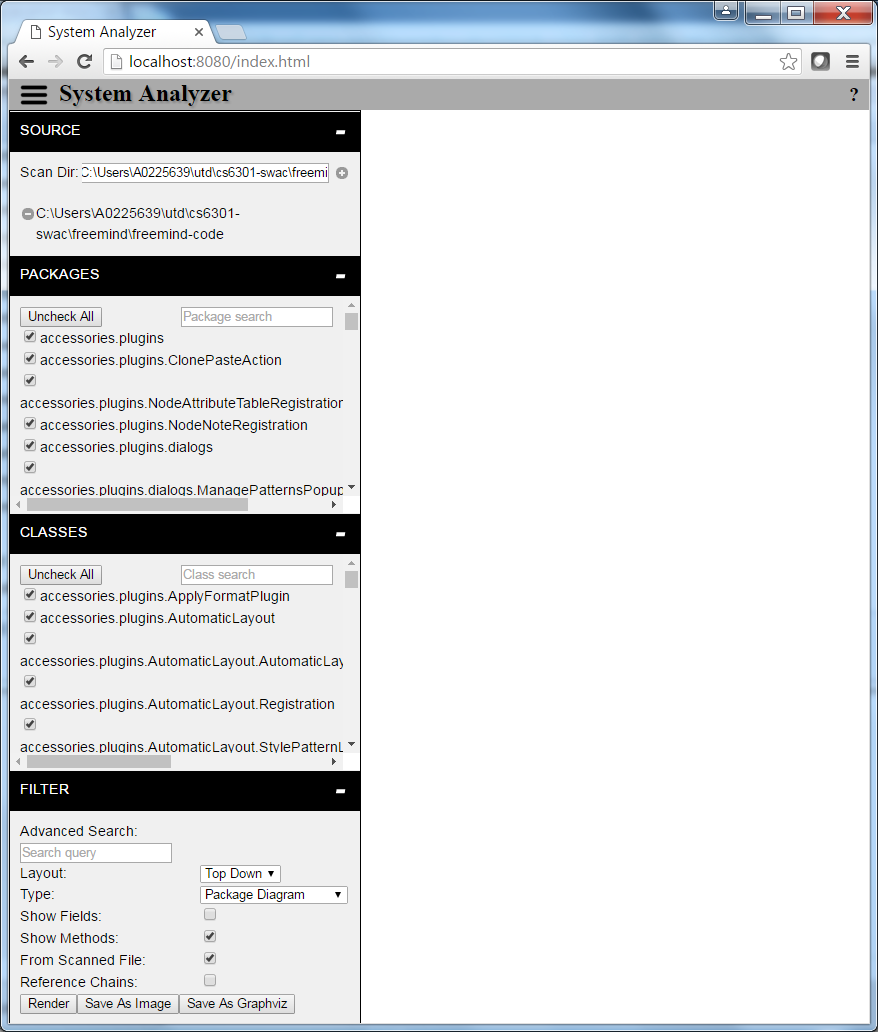
* URLs are removed
* Numeric constants are removed
* CamelCase is split with both the original CamelCase word and split words retained

After the above processing, the following are removed:

* Common English stop words
* Java keywords
* Common Java classes
* Terms of two characters or less

# Vizualizer GUI Description

This section provides a short description of the tool’s GUI and thus further illustrates the features of the tool.



4

3

5

2

1

Figure 1: Vizualizer GUI

## Source Selection

The first part of the GUI is the Source Selection area. This area is labeled with a “1” in Figure 1. This area allows the user to add and remove source folders and files to Vizualizer’s model of a system. Intuitively, the small “plus” icon is used to add to the model and the small “minus” icon is used to remove from the model.

## Package Selection

All of the packages present in the model are listed in the Package Selection area. This area is labeled with a “2” in Figure 1. The user may uncheck any packages they do not want displayed. For convenience, the user is provided with buttons to quickly check and uncheck all displayed packages as well as a text box that can be used to filter the list. When a package is checked, classes that belong to that package are added to the Class Selection section. When unchecked the classes for that package are removed from the Class Selection section.

## Class Selection

This section is the same as the Package Selection section of the GUI but for all the classes of the selected packages in the Package Selection list. This Class Selection area is displayed in Figure 1, labeled with a “3.”

## Filter Specification

The Filter Specification part of the GUI is labeled in Figure 1 with a “4.” The filter specification contains several options, each described below.

### Advanced Search

The Advanced Search field allows a user to enter a query following the Lucene Query Syntax. Only the top ten results from the query are graphed. The results are restricted to packages or classes depending on the type of diagram to be drawn. Note that the Advanced Search does not respect the selections made in the Package and Class Selection fields. Rather, Advanced Search is used as an advanced method to select Packages or Classes. Also, note that more than just the top ten items may be graphed if the Reference Chains feature is used (see Reference Chains below.)

The top ten results are color coded in a yellow hue. The white node will be the top result with the next nine results color coded in shades of yellow that darken as their ranking in the results drops.

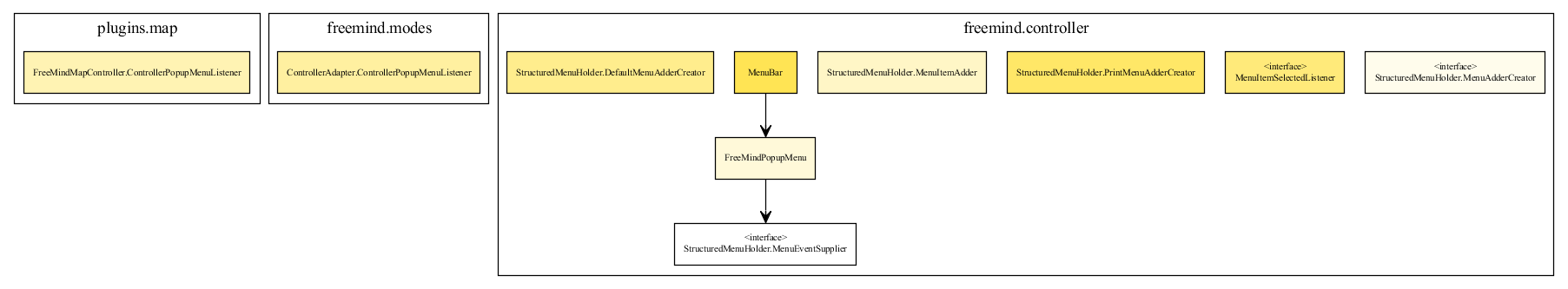


Figure 2: CAD Showing Use of Advanced Search

### Layout

There are two layout choices: Natural and Top Down. Natural uses the Neato layout engine of Graphviz. The figure below shows an example of a Class Association Diagram drawn in Natural layout. Note that in the Natural layout, arrows are collapsed to double headed arrows if appropriate.

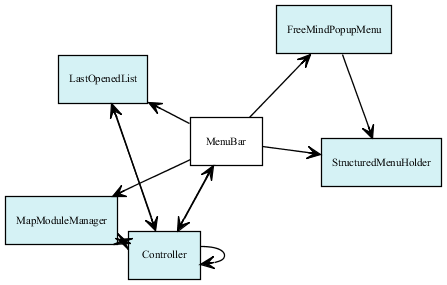


Figure 3: Natural Layout (Color is Part of Reference Chains Feature)

The Top Down selection uses the Dot layout engine of Graphviz. To make Top Down Class Association Diagrams more intuitive, super classes and implemented interfaces are drawn with reversed edges. The implementing class owns the reference in the intermediate model, but the edge is drawn reversed. Thus, when rendered, super classes and implemented interfaces are drawn above the implementing class. Also, note that the Top Down layout draws classes within package nodes, while the Natural layout does not include package information when drawing Class Association Diagrams. An example of a CAD rendered in a Top Down layout is shown in the figure below.

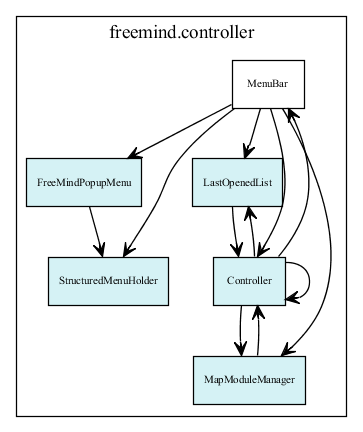


Figure 4: Top Down Layout (Color Is Part of Reference Chains Feature)

### Type

There are three types of graphs. Each is described below.

#### Package

The Package Diagram shows packages as rectangular nodes. Each node contains text which is the name of the package. Directional arrows indicate references from one package to another. A number is drawn next to each arrow to show the number of references made.

References of a package’s classes are counted towards the package’s references. For example, if class APackage.Class makes a call on the class type BPackage.OtherClass, then APackage will have a reference count to BPackage.

#### Class

The Class Association Diagram (CAD) (show in the GUI as “Class Diagram”) shows classes as rectangular nodes. The text within the top most division of the node is the name of the class. Depending on the options specified (see Show Fields and Show Methods below), the fields and methods of the class are drawn within horizontal sub-cells of class node. The fields are listed first and then the methods.

Directional arrows with filled arrow heads indicate a reference from one class to another. A directional arrow with an open arrow head indicates an “extends” relationship. An open directional arrow with a dotted edge indicates an "implements" relationship. The class that extends or implements is located at the tail of the arrow, while the head of the arrow points to the class that is being extended or being implemented.

Any instantiation or use of another class is counted as a reference to that class. In addition, a reference is recorded if the file imports a class.

#### Unreferenced Classes

The Unreferenced Classes Diagram is drawn just as the CAD. However, only those classes that are unreferenced are shown. In determining which classes are unreferenced, super classes and implemented interfaces are not counted toward reference counts in the analysis, but can be shown in the unreferenced classes diagram.

### Show Fields

For the Class Association Diagrams, if checked, fields of the class will be displayed as part of each CAD node. The figure below shows the Controller.LicenseAction class with its single field.

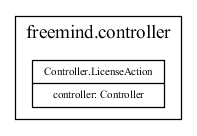


Figure 5: Class Shown with Fields

### Show Methods

For the Class Association Diagrams, if checked, methods of the class will be displayed as part of each CAD node. The figure below shows the Controller.LicenseAction class with its two methods.

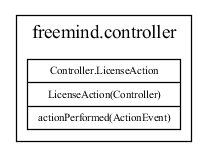


Figure 6: Class Shown with Methods

### From Scanned File

The model built from the source code includes packages and classes not defined within the source code. For example, if a scanned Java source file includes a reference to the java.util.Arrays class, that Arrays class is added to the model. If the “From Scanned File” selection is checked, then such items will not be included in the graph.

### Reference Chains

The Reference Chains feature allows users to make automatic extensions to which nodes are graphed. The list of nodes to graph will begin with the Package list, Class list, or Advanced Search criteria. The nodes will then be extended by the depth of Upstream References and Downstream Dependencies specified by the user. Upstream References can be thought of as in-references and Downstream Dependencies can be thought of as out-references.

The depth and direction at which a node was included is shown by a color coding. The colors darken as their distance from the initial set of nodes increases. Teal hues are used for downstream and blue hues for upstream. If a node was found to be both upstream and downstream from an initial node, it will be colored in a shade of red.

## Render Button

The Render button is used to request a graphing of the system based on all the previously described criteria.

## Save as Image and Save as Graphviz Buttons

These buttons allow the user to download a PNG image or Graphviz (.dot) file of the currently displayed graph.

## Diagram Window

The final section of the GUI is the Diagram Window. This is the 5th area labeled in Figure 1. The requested graph will be drawn in this area of the browser. The Diagram Window supports zoom via the mouse scroll wheel and supports panning using click and drag. In addition, if a class is left-clicked, a context menu will be provided allowing the user to request a view of the source code. The source code for the file containing the class will then be opened in a new browser window.

# Results

## Performance

All performance numbers were collected using a Windows 7 system with 12 GB of RAM and an Intel® Core™ i5-4300U CPU. While collecting the performance numbers, the system was not used concurrently for any other purposes.

### Code Bases Analyzed

The following table provides the list of Code Bases analyzed for the purposes of comparing Vizualizer to other tools.

|  |  |  |
| --- | --- | --- |
| **System Name** | **Repository Tag** | **Link To Repository** |
| FreeMind | 1.1.0 | <http://git.code.sf.net/p/freemind/code> |
| JabRef | v3.2 | <https://github.com/JabRef/jabref.git> |
| Eclipse JDT Core | R4\_4\_2 | <http://git.eclipse.org/gitroot/jdt/eclipse.jdt.core.git> |

Table 1: Code Bases Used For Performance Results

The following table provides metrics for each Code Base. The metrics were collected using SourceMonitor version 3.5.0.306.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **System Name** | **# Source Files** | **# Lines** | **# Statements** | **# Classes** |
| FreeMind | 547 | 116,050 | 49,523 | 1,102 |
| JabRef | 856 | 141,660 | 66,273 | 1,532 |
| Eclipse JDT Core | 6762[[1]](#footnote-2) | 1,659,373 | 602,847 | 8,135 |

Table 2: Code Base Metrics

### Systems Compared

Unless otherwise noted, the default options for the tools were used for all analyses.

|  |  |  |
| --- | --- | --- |
| **Tool Name** | **Version** | **Link** |
| Vizualizer | v0.2 | <https://github.com/CS6301-502-RyanWiles-ErikShreve/vizualizer> |
| Eclipse and  X-Ray | Eclipse 4.5.2  X-Ray 1.0.4.1 | <https://www.eclipse.org/>  <http://xray.inf.usi.ch/> |
| Understand | Build 838 | <https://scitools.com/> |

Table 3: Details of Systems Compared for Performance

### Parsing Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Base** | **Vizualizer** | **Eclipse and X-Ray** | **Understand** |
| FreeMind | 17 s | 32 s  (6 s + 26 s) | 9 s |
| JabRef | 22 s | 44 s  (22 s + 20 s) | 1 m 47 s |
| Eclipse JDT Core | 5 m 44 s | 3 m 40 s  (37 s + 3 m 3 s) | 5 m 37 s |

Table 4: Parsing Performance Results

The performance of Vizualizer matches well to the other systems.

The Eclipse and X-Ray tool had good performance across all three code bases. The total time is listed along with the time Eclipse took to parse the source and then the time X-Ray took to analyze the project. We do note that getting the code bases into an Eclipse project that the X-Ray tool would parse was difficult as none of the code bases are delivered with Eclipse project files. Freemind is built using Ant, JabRef is built using Gradle, and Eclipse JDT Core is built using Maven. X-Ray would not parse projects created in Eclipse by importing Gradle or Maven build scripts. Thus, the Eclipse projects were built by creating source folder links to each individual src folder in the respective code base.

Understand had much better per-line performance for Eclipse JDT Core than expected given the FreeMind and JabRef numbers. One possible reason is that Understand reported 2100 parse errors when parsing JDT. It is possible that Understand performed less analysis due to these errors. The analysis was repeated with similar numbers obtained.

### Graphing Results

The graphing performance was taken for each Code Base using the tools shown below.

For Vizualizer, the tool was requested to create a Class Association Diagram of all parsed classes, without methods or fields displayed. It was also requested to present a Package dependency diagram, again with all parsed packages displayed. The Top Down layout was used for both diagrams. Class Association Diagrams were drawn without fields or methods shown. Only elements from parsed files were included in the graphs. The browser used was Chrome 50.0.2661.94 m (64-bit).

We do not compare the graphing time to other tools since we were unable to locate any that provide the same graph as Vizualizer. The closest found was the commercial tool Understand. A trial version was downloaded for comparison purposes. Understand presents a few different dependency graphs. However, the Understand graphs only show the immediate dependencies of a selected object. To get a full dependency view as in Vizualizer, one has to repeatedly request Understand to expand its graph nodes. Understand’s graph also lacks the ability to show loops in the dependency, instead drawing a tree that is unending as nodes are repeated (the graph is a tree structure).

|  |  |  |
| --- | --- | --- |
| **Code Base** | **Vizualizer Class Association Diagram** | **Vizualizer Package Dependency Diagram** |
| FreeMind | 1 m 30 s | 14 s |
| JabRef | 3m 15 s | 4 m 41 s |
| Eclipse JDT Core | > 1 hour.  Test stopped. | > 1 hour.  Test stopped. |

Table 5: Graphing Performance Results

The generating of graphs for the Eclipse JDT Core system took an unreasonable amount of time and thus the tests were stopped after one hour of run time.

## Graphs

This section provides and describes example graphs produced by the Vizualizer tool.

The diagram in Figure 7, shows the tool’s Class Association Diagram for JabRef when the Advanced Search query “search” is used, with depth of ‘1’ Upstream References enabled. This graph represents a starting point for understanding how JabRef’s search functionality is implemented. The top ten classes returned for the query are shaded in yellow. As can be seen, not all of the classes returned from the query are connected directly. However, by adding the Upstream References feature, the way the system connects these classes can be seen.

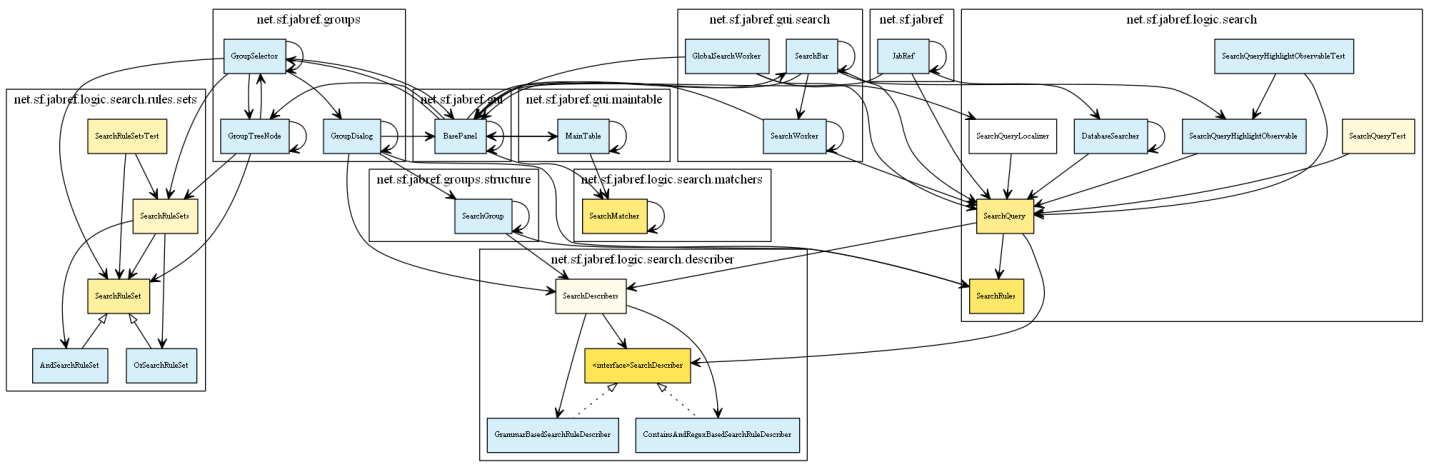


Figure 7: JabRef CAD for "Search"

In Figure 8, below the Class Association Diagram of FreeMind with the Advanced Search query of “Menu” is shown. Note the minimal number of connections. This may represent that the implementation of the menu concept is spread out within the code base and thus may indicate an opportunity for refactoring to increase maintainability.

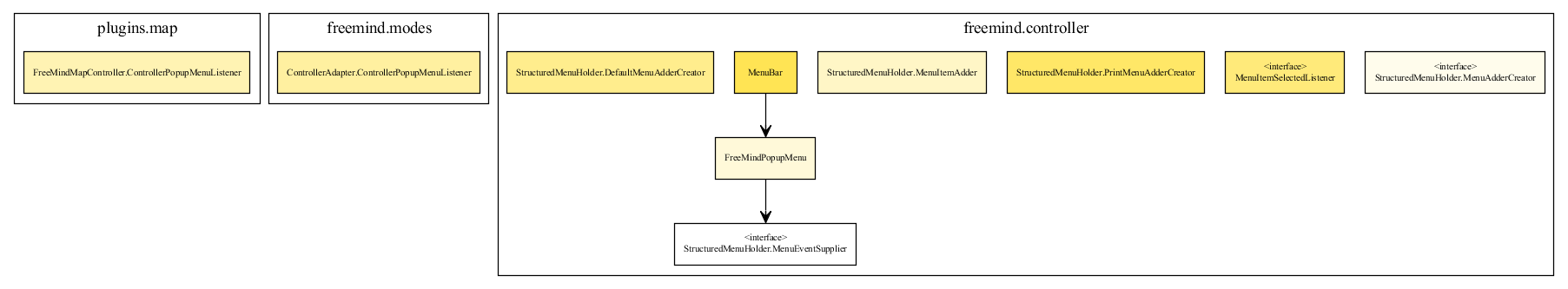
****

Figure 8: FreeMind CAD for "Menu"

# Future Work

## Improvement of the Graphing Engine

The time taken by the JavaScript implementation of Graphviz is likely too long for usefulness. Developers are likely accustomed to receiving instant results when exploring a code base. Alternate graphing libraries, or performing the graph generation outside the browser, may result in a better user experience.

## Include a Metrics Sidebar

A sidebar with various metrics could be included in the UI. The metrics shown would relate to the item the mouse cursor is hovering over. Various code quality metrics such as cyclomatic complexity, LLOC, nesting level, and other quality metrics could be included.

## Search Improvements

Instead of adding all text to a single Lucene field, multiple fields could be indexed in Lucene. For example, comments, methods, class names, fields, and filenames could all be indexed as different fields. This would allow the user to use Lucene’s query language to specify searches within the specific fields. This would be useful for the cases were the user has a more specific item they are looking for. For example, they may want to find all classes that contain a method of a specific name.

## Database Integration

The model can be extended to include a representation of databases. This would allow the tool to identify relationships between classes and database tables and fields.

## Watch Functionality

The tool can be extended to watch the files added to its model. Upon detecting a change to the file, the model can be updated with the new file contents and the graph updated without requiring any input from the user.

## UI Improvements

There are several things that could be improved about the UI. These include:

* Package and class lists could be shown in tree views
* Context menu could include ability to remove a node from the graph
* Help screens could be added
* Color Legend could be added
* The graphing colors could be made configurable

# Conclusion

We have developed and presented a tool combining source code visualization and text retrieval to aid developer understanding of Java code bases. Our tool parses Java source files into an internal model and then presents to the user various diagrams to aid in understanding of how the classes and packages of the Java source relate to each other and to classes and packages outside the source.

Our tool has reasonably fast performance and places a minimum burden on the developer prior to generating graphs (the tool does not require a buildable project.) While the tool provides a foundation for continued development, we believe it already presents usefulness to developers in its current state via the following two key features:

* Search based selection of Packages/Classes to graph and
* Extension of selected Packages/Classes using Reference Chain Depths.

# Appendix A: Survey of Other Tools

As part of our project, we surveyed the availability of other software engineering visualization tools. This list is not limited to only source code visualization (for example, tools to visualize project commit history are included). These tools include commercial and open source tools and cover a wide range of features and capabilities. Because we could not find a decent list of visualization tools already constructed, we have provided our list here along with short notes on the capabilities of the tools. Of course, the list could be expanded even further. Note that we have not used most of these tools and our notes are based on the publically available literature for the tool.

| **Name** | **Min. Cost [[2]](#footnote-3)(OSS=free)** | **Language Support** | **Visualizations Provided** | **Last Release[[3]](#footnote-4)**  (DD-MM-YYYY) | **Notes** |
| --- | --- | --- | --- | --- | --- |
| BoUML | $57 | C, C++, Idl, Java, MySQL, PHP, Python | UML | 18-03-2016 | Tool’s focus appears to be more on UML drawing and UML to source conversion. However, some source to UML conversion is supported. |
| Code City | OSS | Requires MSE model of system. | Code City with dependency/reference overlays | 18-11-2009 | MSE is a system model format. Tools exist to create models for Java, C++, C#, and SmallTalk. However, it appears most of them are not maintained well.  Code City was a research project with several papers published. |
| Code City Eclipse Plugin | Unknown (free for personal use) | Java | Code City (no dependency/reference overlays) | 02-22-2016 | Not free for commercial use, but no price listed. |
| code\_swarm | OSS | Subversion, CVS, Git, Mercurial, Perforce, VSS, Starteam, Wikiswarm, Darcs | Commit-to-Developer Animation | 24-07-2011 | Animates commit history of a project. |
| CodeFlower | OSS | User provides data | Flower | 07-11-2014 | Graph just shows directory structure with node size determined by user defined metric. User must create data for graph themselves. |
| CodeGraph | OSS | C, Java, PHP, possibly others (documentation unclear) | Dependency Structural Matrix | 17-04-2015 | Released by Facebook. |
| CodeViz | OSS | C, C++ | Call Graph | 27-04-2013 | Last release date based on release’s tar time stamp. |
| Dependency Visualizer | OSS | .NET Projects | Project Dependency Graph | 14-11-2008 | Maps dependencies between projects. |
| DependencyWheel | OSS | User provides data | Wheel | 23-02-2016 | Graph connects nodes in a wheel pattern, with thickness of connections representing strength of dependency. User must create data for graph themselves. |
| DoUML | OSS | C++, Idl, Java, PHP, Python | UML | 05-09-2015 | Fork of BoUML prior to BoUML going commercial. |
| Gource | OSS | Subversion, CVS, Git, Mercurial, Bazaar | Developer-to-Commit Animation | 02-02-2016 | Animates commit history of a project. |
| kcachegrind | OSS | Data from Cachegrind and CallTree tools | Call Graph, Treemap of Cost, Caller Map, Callee Map | 5-4-2013 | Provides information from runtime profiling data. |
| Linguine Maps | OSS | XML based files (like ANT build files) | ER Diagrams | 13-03-2013 | None. |
| nDepend | $339 | .NET Languages | Metric Trend Graph, Metric Tree Map, Dependency Matrix, Dependency Graph | 29-04-2016 | None. |
| PDE Incubator Dependency Visualization | OSS | Java (Eclipse Plugins) | Caller and Callee trees | 22-02-2016 | Eclipse plugin for graphing dependencies of Eclipse plugins (not of general source). |
| SciTools Understand | $995 | Ada, C, C++, C#, FORTRAN, Java, JavaScript Jovial, Delphi/Pascal, PL/M, VHDL, Cobol, PHP, Python | Metric Tree Map, UML Class Diagram,  Control Flow Graph, Hierarchy Graph, Declaration Graph, Dependency Graph | 22-04-2016 | Billed as a Static Analysis Tool and includes an editor. |
| Sonargraph-Explorer | Free | C++(?)[[4]](#footnote-5), C#, Java | Dependency Graph, Cycle View, Hierarchical Dependency View | 30-11-2015 | Tool is free but requires registration. |
| Sourcecode Visualizer | OSS | Java | Call graph | 22-02-2016 | Eclipse plugin. |
| X-Ray | OSS | Java | System Complexity,  Class Dependency View, Package Dependency View | 03-08-2013 | Eclipse plugin. |

Table 6: Tool Survey

| **Name** | **Web Link** |
| --- | --- |
| BoUML | <http://www.bouml.fr/> |
| Code City | <http://wettel.github.io/codecity.html> |
| Code City Eclipse Plugin | <https://marketplace.eclipse.org/content/codecity> |
| code\_swarm | <http://www.michaelogawa.com/code_swarm/> |
| CodeFlower | <https://github.com/fzaninotto/CodeFlower> |
| CodeGraph | <https://github.com/facebook/pfff/wiki/CodeGraph> |
| CodeViz | <http://www.csn.ul.ie/~mel/projects/codeviz/> |
| Dependency Visualizer | <http://dependencyvisualizer.codeplex.com/> |
| DependencyWheel | <https://github.com/fzaninotto/DependencyWheel> |
| DoUML | <https://github.com/DoUML/douml> |
| Gource | <http://gource.io/> |
| kcachegrind | <https://kcachegrind.github.io/html/Home.html> |
| Linguine Maps | <http://www.softwaresecretweapons.com/jspwiki/linguinemaps> |
| nDepend | <http://www.ndepend.com/> |
| PDE Incubator Dependency Visualization | <https://marketplace.eclipse.org/content/pde-incubator-dependency-visualization> |
| SciTools Understand | <https://scitools.com/> |
| Sonargraph-Explorer | <https://www.hello2morrow.com/products/sonargraph/explorer> |
| Sourcecode Visualizer | <https://marketplace.eclipse.org/content/sourcecode-visualizer> |
| X-Ray | <https://marketplace.eclipse.org/content/x-ray-software-visualization> |

Table 7: Tool Survey Link

1. About 60 files are not included in metrics due to SourceMonitor parse errors. Most of these files were test files intended to test JDT’s ability to parse Java code. The collected statistics still suffice to show the general size of the code base. [↑](#footnote-ref-2)
2. The minimum cost to support a single developer in a professional setting. In most cases this would be a “named license” for use by an employee of a business (i.e. tool is not for “personal use.”) These prices are in US dollars, are based on information of the tool vendor’s website, conversion rates from xe.com, and were collected on April 29th, 2016. [↑](#footnote-ref-3)
3. As of April 29th, 2016 [↑](#footnote-ref-4)
4. Conflicting information on C++ support was found. [↑](#footnote-ref-5)