USE OF 3D PROGRAM VISUALIZATION TO SHOW VISIBILITY, COHESION, AND QUALITY OF JAVA CLASS ELEMENTS

By

ANDREA GOETHALS

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

ACKNOWLEDGMENTS

I would first like to thank my husband and personal system administrator, Wesley, for not only being the sole ‘breadwinner’ in the family for the last seven months, but also for being incredibly helpful and always a good listener.

I would also like to thank my chair, Paul Fishwick, for proving to me that there is a place for art and creativity in computer science, and for allowing me to explore my research interests while always being available to guide me in the right direction.

TABLE OF CONTENTS

page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

USE OF 3D PROGRAM VISUALIZATION TO SHOW
VISIBILITY, COHESION, AND QUALITY OF JAVA CLASS ELEMENTS

By

ANDREA GOETHALS

May 2002

Chair: Paul A. Fishwick
Major Department: Computer and Information Science and Engineering

JVisualizer is a prototype application created to help beginning Java programmers understand the structure, design and coding habits of their Java programs, and which parts of their programs need the most improvement. JVisualizer uses common object-oriented and Java-specific programming guidelines to critique input Java programs, and generates a 3D visualization of the program, as well as a detailed report on the program.

The visualization was designed in a way that does not produce a high cognitive load, by using visualization presentation techniques such as color, size, shape, position and visual metaphors. The user can interact with the visualization to get more information about their program and identify the lowest-quality elements in the program. The generated report tells the programmer why particular elements of his/her program are of low-quality. The long-term goal of JVisualizer, as well as applications that can be modeled after JVisualizer, is that it can train programmers to write programs that are consistent with programming guidelines, resulting in programs that are easier to understand and maintain.

CHAPTER ONE
INTRODUCTION

Motivation

Software maintenance accounts for the largest percentage of the total cost of software products [Sommerville, 2001]. In addition, software continues to get more complex [Bassil and Keller, 2001]. For these reasons it is imperative that beginning programmers are taught how to design and write programs that will be easy to understand and maintain. Tools that can help the programmers visualize their code and pinpoint design and coding problems in the code can assist in this training.

There are many published guidelines for designing and coding Java programs so that the programs are relatively easy to understand, maintain and modify [Ambler, 2000; Badeaux, 1999; Coad and Mayfield, 1999, etc.]. There are also quantitative metrics for analyzing the "quality" of Java and other object-oriented programs [Lorenz and Kidd, 1984; Henderson-Sellers, 1996]. Software applications such as Together/J from TogetherSoft can output 2D graphical charts showing the result of individual metrics run on a user's source code (see Figure 1).

What are not currently available to beginning programmers are tools that let the programmer visualize their programs as a whole. Most software visualization (SV) is 2D and can not display the results of multiple analyses of the code. In addition, most SV tools do not provide constructive criticism about the quality of the code. There are not tools that show programmers the parts of their programs that need the most attention, or

that reinforce their good programming habits.

This research explores how visualization techniques, 3D graphics, and visual perception theory, such as the Gestaltian laws of grouping, can be used to create intuitive graphics showing the quality of a Java program and its sub-class elements. These visualizations isplay the result of comparing the programmer's source-code program with selected programming guidelines. The guidelines were chosen so that they reflect the most common general object-oriented programming guidelines, as well as some specific to Java programs. The selected guidelines also relate to the maintainability of the code.

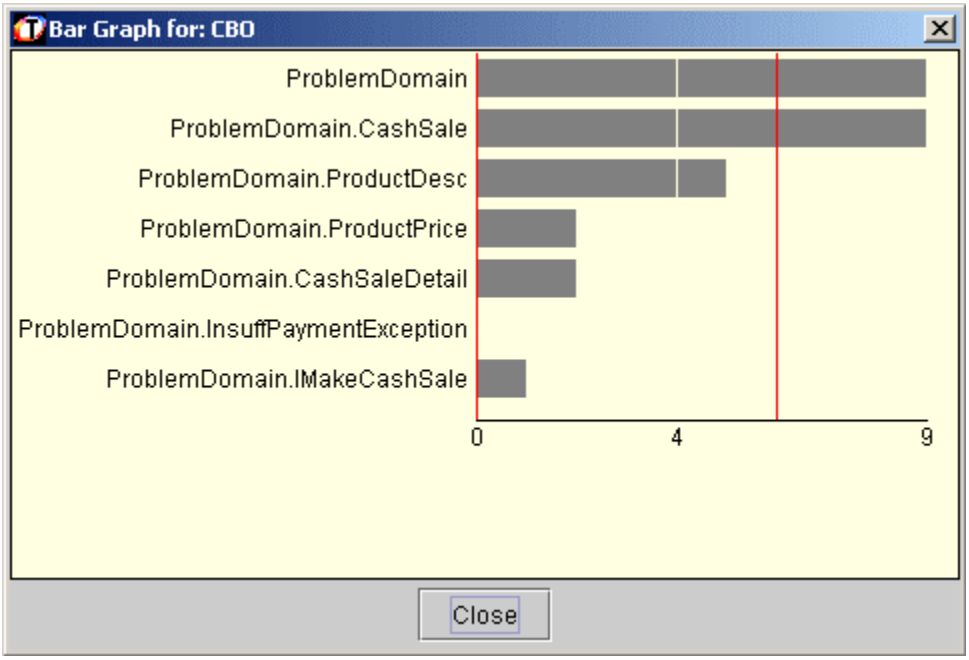

Figure 1-1: A bar chart produced by TogetherSoft's Together/J showing the result of the CBO (Coupling Between Objects) metric

The Research Problem

The main problem this research addresses is how to illustrate the quality of a Java class and its members in a way that is efficient and easy to understand. 'Quality' is defined as the degree of compliance with common object-oriented and Java-specific programming guidelines. By 'efficient' it is meant that the results of many tests can be

shown in the same graphic. By 'easy to understand' it is meant that the graphic does not demand a high cognitive load. [Kraemer et al. 2001] note that the high cognitive load of visualizations lead to them not being used in practice as much as they could be.

Sub-problems of the research include the problems of how to organize the structure of the class and its members, how to show that some class members are visible (within the same scope) and some are not, and how to illustrate the degree of coupling between the class' methods and variables. The target audience of this research is beginning Java programmers who have at least a rudimentary knowledge of Java programming and who possess a reasonable ability to abstract.

CHAPTER 2
PROGRAM VISUALIZATION

Overview of Software and Program Visualization

Program Visualization is a sub-field within the larger field of Software Visualization (SV), which also includes visual programming, algorithm animation and information visualization. Software visualization is defined by (Price et al. 1993) as the use of "the crafts of typography, graphic design, animation, cinematography and modern human-computer technology to facilitate the human understanding and effective use of computer programs". These sub-fields of SV share the characteristic that they are focused on translating, filtering, and presenting large amounts of data into a format that is easily digestible to humans.

The earliest forms of SV were automated flowcharts of assembly language and Fortran programs developed by Haibt in 1959 (Haibt 1959). Animated algorithms showing sequential, and later concurrent, execution were common in the 1960s, '70s and '80s. The 1980s saw the development of high-level SV tools for functional programming languages, like Pascal. It was not until the 1990's that researchers began to study the unique SV requirements of object-oriented programs.

SV has been used for many purposes, including aiding software developers with software reengineering, development, understanding, testing and performance tuning; teaching algorithms and data structures; and data modeling. Another purpose of existing SV tools is to improve the 'navigability' of software (Storey et al. 1998) or the WWW.

The role of SV tools up to now have been to describe the nature of the program, algorithm or data structure 'as they are'. They have not been used to explicitly suggest improvements to these structures.

Program visualization is the visualization of actual program code or data structure in either dynamic or static form to enhance the human understanding of the program. The first OOP program visualization tools depicted the static structure of the code, and included class browsers, inheritance viewers, affinity browsers, breakpoint debugging and object inspection (De Pauw et al. 1993). More recent research has focused almost exclusively on the dynamic structure of the program in execution, for example on the instantiation of objects and the passing of messages between objects.

(Kraemer et al. 2001) lists three characteristics that a program visualization tool should have: expressiveness, visual classification, and continuity. Expressiveness means that the visualization should show only those elements that are of interest to the user and nothing else. Visual classification means that the visualization should distinguish between different program entities. Continuity means that for the duration of the visualization session, related data should be depicted in a common graphical way, and unrelated data should be depicted using dissimilar graphical features.

Visualization Presentation Techniques

There are many different presentation techniques used in the display of visualizations. These include the use of filtering, focusing, sampling, linking, context, three spatial dimensions, color, and visual metaphors. Filtering means to eliminate or minimize information, or "noise", that is not of importance or interest in the visualization (Catton and Murphy 2001; Maletic et al. 2001). Focusing is used to draw attention to

elements of interest. Sampling means to select particular elements or events of the algorithm, program or system to animate. Linking means to highlight, manipulate or edit elements in multiple visualization views concurrently where they are views of the same elements (Graham and Kennedy 2001). Context techniques show overall views of the data, even when the visualization is 'zoomed in' to a smaller scale. The presentation techniques of three spatial dimensions, color, and visual metaphors are described in more detail in the following sections.

Three Spatial Dimensions

Early software visualization was entirely 2D, in part because of hardware limitations. Because of the current prevalence of inexpensive, high-performing hardware, 3D visualizations are possible. Many researchers have noted both benefits and drawbacks of using 3D for program visualization.

3D visualizations are desirable for some applications because the data is inherently 3D (Stasko 1992). For non-3D data, 3D visualizations provide an extra dimension with which to encode more information about the data (Stasko 1992). May Cheng (Cheng 1998) presents four additional benefits of 3D visualizations. They allow for new perspectives on data usually seen in 2D. They make it possible to display relationships or connections between multiple 2D views. The extra dimension can be used to display history, and finally, if nothing else, the extra dimension can be used for aesthetics. Some experiments have shown that 3D visualizations lessen the cognitive load as compared to their 2D counterparts (Maletic et al. 2001), although there are many examples of 3D visualization tools that place a high cognitive load on the user.

Drawbacks of using 3D for program visualizations are listed by Cheng (Cheng 1998). There are rendering delays caused by the complex objects and animations used by 3D. Because it is usually possible to change the viewing perspective in a 3D visualization, the visualization author can not assume that the user will be viewing it from a particular view perspective. The additional degree of freedom means that there are more choices to make as to how to map the data to the degrees. Finally, the system has more object properties to store in 3D, for example the 'z' values. It must be noted that the existence of modern high-performance computers lessen the first and last drawbacks listed by Cheng (Cheng 1998). The second and third drawbacks, the viewing perspective and the complicated design problems, remain as valid criticisms.

The three spatial dimensions of 3D visualizations can each be used to show one of five elements (Stasko 1992). These elements are: value, position, state, history and aesthetics. A value dimension can be used to encode the contents of a variable. A scaling factor or mapping from the variable's value to the graphical screen would need to be established. A position dimension can represent an ordering of the data, for example, an index into a set, or a row within a set of records. Positional dimensions are nominal or ordinal data. The state of a visualization is usually stored not in a spatial dimension, but in the time dimension. The history of a visualization can be stored in a spatial dimension. For an animation, the history would include events that have already occurred. The last element, aesthetics, can also take up a spatial dimension. For example, the 'z' value can be a constant value for a dataset just to give an otherwise 2D visualization more appeal.

Computational visualizations in 3D are grouped into 3 categories by Stasko (Stasko 1992), depending on the minimal number of spatial dimensions needed for the

data used for the visualization, and what element each of the three spatial dimensions shows. These three categories are summarized in Table 2-1.

Table 2-1: 3D Computational Visualization Categories

Type of 3D Computational Visualization	Nature of Data	Use of Spatial Dimensions
Augmented 2D	Minimally requires 2 spatial dimensions	Dimension 1: position 1 Dimension 2: value 1 Dimension 3: aesthetics
Adapted 2D	Minimally requires 2 spatial dimensions	Dimension 1: position 1 Dimension 2: value 1 Dimension 3: value 2 or Dimension 1: position 1 Dimension 2: position 2 Dimension 3: value 1 or history
Inherent 3D Application Domain	Minimally requires 3 spatial dimensions	Dimension 1: value 1 and possibly position 1 Dimension 2: value 2 and possibly position 2 Dimension 3: value 3 and possibly position 3

Examples of the augmented 2D visualization are the 3D bar charts popular in newspapers. These charts do not add any more information to the visualization by using 3D than if they used 2D. An example of an augmented 2D visualization is shown in Figure 2-1. The x axis shows a value, the y axis shows the position within the year and within a hardware category, and the z axis is used for aesthetics.

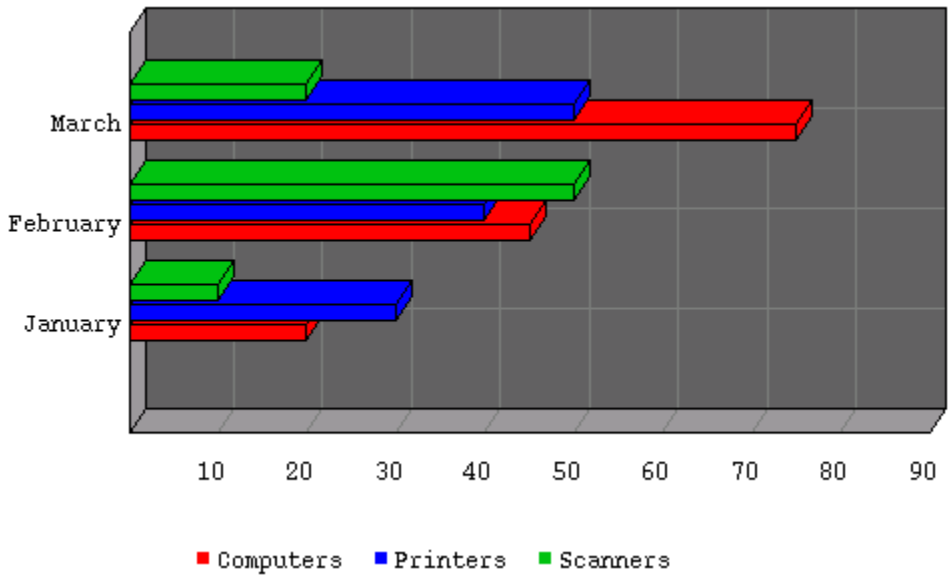

Figure 2-1: An example of an augmented 2D visualization
Source: Harisan 2001.

An adapted 2D visualization adds additional information with the extra spatial dimension. This type of visualization can show innovative views of the data. An example of this type of visualization is shown in Figure 2-2. A class hierarchy in plan view is shown in the larger part of the figure, with an elevation view inset in the lower right of the figure. The x and y dimensions are used to show position within the class hierarchy. The z dimension is used to show the values of aspects of each method within each class.

An inherent 3D application domain visualization is used to visualize data that can not be analyzed in less than three spatial dimensions. Examples of data that would fall in this category are non-planar graph algorithms, volume packing data, and cube parallel architecture (Stasko 1992). An example of this type of visualization is shown in Figure 2-3. The x, y and z dimensions are used to show position within a particular sphere and within the cube, as well as value of the size of the spheres.

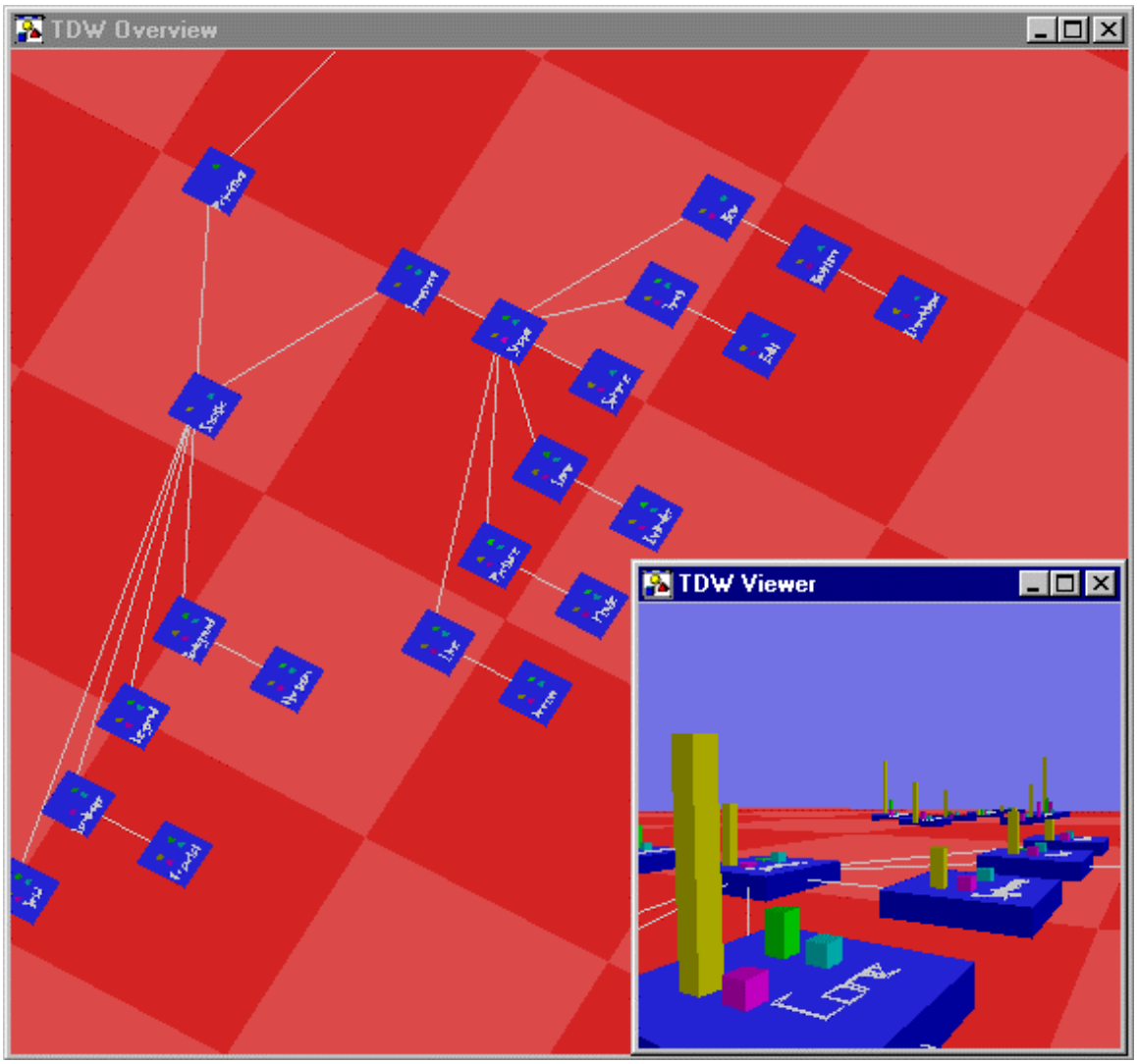

Figure 2-2: An example of an adapted 2D visualization
Source: Mitchell 2001.

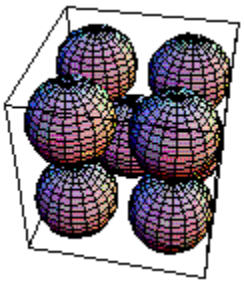

Figure 2-3: An example of an inherent 3D application domain visualization
Source: Friedman 2001.

Visualizations can be moved from two spatial dimensions to three by several techniques (Cheng 1998). The extra dimension can extend another property of the data, or it can represent time. Alternatively, the extra spatial dimension can be used to integrate multiple 2D visualizations. Finally, the extra dimension can be saved until needed for animation, for example for zooming in or out.

Color

A second technique used in many, but perhaps not enough (Price et al. 1993), visualizations, is color. Color can be used to provide additional information without adding to the cognitive load of the visualization (Price et al. 1993). It can be good for displaying the state of data structures or algorithms, highlighting areas of interest, uniting multiple elements, emphasizing patterns and elements, and capturing history (Cheng 1998, Price et al. 1993). Colors can be used to represent magnitude by choosing a color scale, such as the spectrum, grayscale shading, or other scales (Spence 2001).

Color has three different "dimensions": hue, saturation and lightness. Hue is the color's wavelength. Saturation is the color's grayness, or how pure it is. Lightness, also known as value or brightness, is its shade. Figure 2-4 shows the difference between these three dimensions. Edsall (Edsall 1999) describes how each dimension can be used in visualizations. Hue differences can best be used to differentiate between nominal types of data. Lightness can best be used to differentiate between ordinal data. It is not known how saturation can be used for visualization.

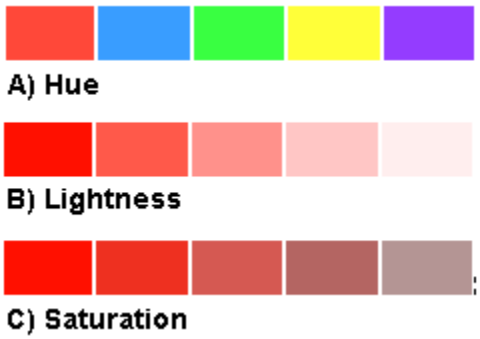

Figure 2-4: Differences in Hue, Lightness, and Saturation
Source: Edsall 1999.

Several authors caution about the potential to misuse color in visualizations (Cheng 1998, Spence 2001). Humans can successfully identify meanings of only a small number, five to six (Cheng 1998), of colors at one time. Too many colors displayed add to the cognitive load of the visualization. There is no objective ordering that can be conveyed by colors. It is technically difficult to include either a legend or labels along with a visualization. Additionally, six to eight percent of males have at least a mild form of color blindness (Foley et al. 1997).

Visual Metaphors

In the context of visualizations, visual metaphors are mappings between the data being visualized and graphical elements of the visualization. Visualizations can take many different formats, but there are some visual metaphors that are used often. These metaphors include bar charts, pie charts, matrix views, trees, landscapes, cityscapes, network views, scatterplots, histograms, data sheets, ParaBoxes, and time tables (Eick and Karr, 2000; Eick et al. 2000).

Visualization Design

The design of a visualization affects how effective and easy to use it will be. (Foley et al. 1997) lists three elements to consider when designing a user-computer

interface which also apply to visualizations: visual clarity, visual codings, and good layout principles.

Visual Clarity

Visual clarity can be used to reinforce and emphasize the underlying logical organization of the data. The Gestalt rules, used by graphic designers (Foley et al. 1997) are a means to visual clarity, with the objective being to minimize the physical and cognitive load on the user's acquiring information from the presentation. The main idea behind these rules is that the way in which elements are grouped affects our perception of how these elements relate to each other.

There are four Gestalt rules or laws of grouping: similarity, proximity, closure, and simplicity. The law of similarity states that elements that are graphically similar tend to be seen as related. The law of proximity states that elements that are geographically closer to each other are seen as related. The law of closure states that if elements form a pattern or enclose an area they will tend to be seen as related. The law of simplicity states that elements will tend to be seen as related if they form a symmetric, smooth or regular form when grouped together.

Visual Codings

The visual coding, or graphical vocabulary of the visualization, can be used to distinguish between the elements of the visualization. The techniques that can be used are: color, shape, size or length, typeface, orientation, intensity, texture, line width, line style, and gray level (Foley et al. 1997, Price et al. 1993). For each of these techniques there is a maximum number of variations of the technique that can be used before the user stops being able to recognize the variations. Foley et al. (Foley et al. 1997) list the

maximum number of variations as shown in Table 2-2.

Table 2-2: Maximum number of variations of presentation techniques discernible for a 95-percent error-free performance

Technique	Maximum Number of Variations
Different colors	10
Different area sizes	6
Different lengths	6
Different intensities	4
Different angles	24
Different geometric shapes	15

Source: Foley et al. 1997.

The visual coding should be appropriate for the type of data being depicted (Foley et al. 1997). For nominative data, color is better to use than size, intensity or shape. For ordinal data, coding with an obvious ordering, like line styles and area-fill patterns with varying densities should be used. For ratio data, a coding that can vary continuously should be used. The most accurately recognized coding, in order of best to worst, is shown in Table 2-3 (Foley et al. 1997).

Table 2-3: Relative effectiveness of techniques to use for ratio data

Technique	Relative Effectiveness (sorted most to least effective)
Position along a common scale	Most Effective
Position on identical, nonaligned scales	
Length	
Angle between two lines, and line slope	
Area	
Volume, density, and color saturation	
Color hue	Least Effective

Source: Foley et al. 1997.

To attract a user's attention to a particular element of the visualization, there are several techniques that can be used. A unique color or shape, or a blinking, pulsating or rotating object can be used. A unique color is more effective for this purpose than the other techniques (Foley et al. 1997).

Good Layout Principles

In order to hold the interest of the user and to control the attention of the user, the visualization should conform to good layout principles. The aesthetic elements that have a positive effect on the user are: repetition, similarity, equality, contrast, symmetry, balance, center of focus, and sequence (Flood and Carson 1993, Foley et al. 1997). The elements that have a negative effect on the user are ambiguity, undue repetition, and unnecessary imperfection (Flood and Carson 1993).

CHAPTER 3
JAVA CLASSES

Hierarchy and Organization among Java Classes

The Java class hierarchy contains all Java classes, whether developed by Sun or third party programmers. The class hierarchy is a single inheritance tree with the Object class at the root, and all other classes as nodes within the tree. All classes that are not explicitly declared as subclasses of a Java class other than Object are children of Object and inherit functionality from Object. Classes that are declared as a subclass of a Java class other than Object inherit all of Object's static methods as well as all non-static non-overridden methods of Object. All classes that could be developed by a Java programmer are subclasses of some class.

Java interfaces are templates for creating classes. They contain abstract methods without implementations that must be implemented by classes implementing the interface. Unlike the situation with class inheritance, where a class can have only a single superclass, a class can implement multiple interfaces.

Classes and/or interfaces that are related to each other can optionally be organized into packages. Packages provide a way to restrict access to class data and functionality as well as being an organizational tool.

Beginning Java programmers typically learn Java by starting at the level of the class. They do not learn how to design multi-class packages or to use interfaces until later in their careers. For this reason, the scope of this research is at the level of a Java class,

including elements within a Java class that are both visible and invisible from outside the class. This precludes the incorporation of Java elements and object-oriented concepts external to a single Java class such as inheritance, packages, and interfaces in this research. Figure 3-1 shows the Java elements that are not included in this research.

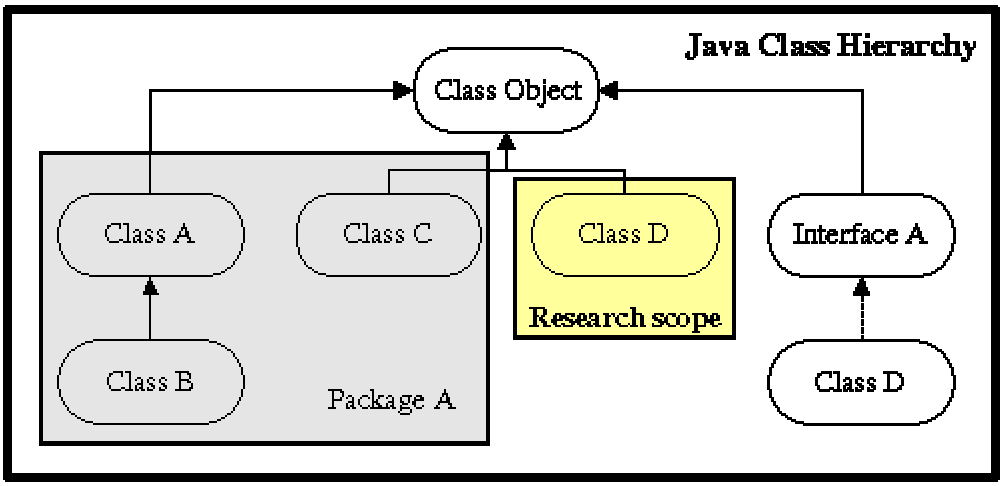

Figure 3-1: Scope of Research
Notes: The Java Class Hierarchy is a tree data structure with the class Object at its root. All solid arrow lines represent inheritance "extends" relationships. For example, class B extends class A, which extends class Object. The dotted arrow line between class D and interface A represents an "implements" relationship. The scope of this research is at the level of a single class and does not include any class or interface inheritance relationships, nor does it include package elements.

Java Sub-Class Elements

The scope of this research is at the level of the Java class and includes many elements contained within Java classes. These elements vary in their type, where they can be found within a Java class, and in their scope or visibility. The following sections describe the organization of these sub-class elements and the scope of their visibility and accessibility.

Element Organization

Data and functionality. At a very high level of abstraction, the two main elements encapsulated by Java classes are data and functionality. Some of the data and functionality belong to objects created at runtime, other data and functionality belong to the class itself. The data are called variables, and the functionality is defined in methods. An object's variables are called instance variables or fields, and its methods are called instance methods, or just methods. A special type of instance method is called a constructor, because it is used to construct the object. The class' variables are called class variables or static fields. The class' methods are also known as static methods. All of this information is summarized in Figure 3-2. Class variables and methods are recognizable by the modifying keyword 'static'. Classes may also contain inner classes and interfaces, but this research will not cover these elements.

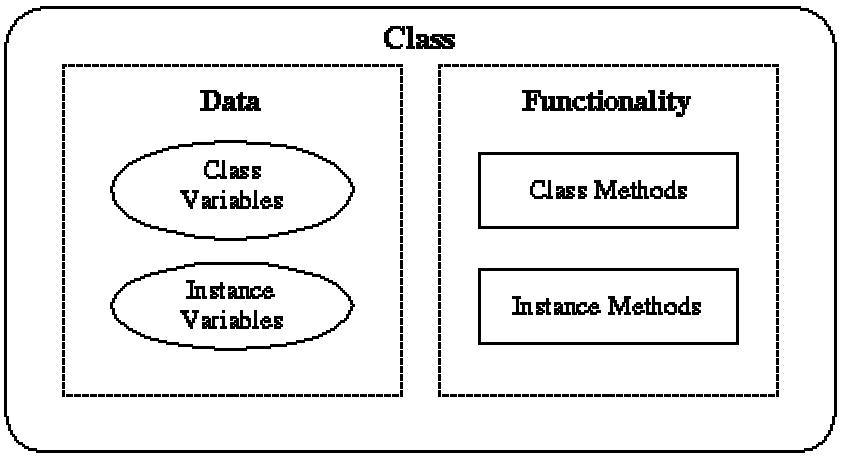

Figure 3-2: The primary elements within a Java class: class variables, instance variables, class methods, and instance methods

Variables. In addition to class and instance variables, there are four other types of variables found in Java classes (Daconta et al. 2000). These include method parameters, constructor parameters, exception-handler parameters, and local variables. All four of

these variable types are associated with instance methods; class methods can contain any but the constructor parameters. A local variable can be declared within a block or within a "for" statement, both of which are contained within methods. There are three special types of local variables: streams, loop counters, and exceptions. A class, instance or local variable can be a declared constant by using the "static final" modifier keywords, although they are typically class variables. Figure 3-3 shows where each type of variable is found within a Java program, specifically whether or not they are located within or outside of methods. Note that class and instance methods are composed of two parts: the header and the body.

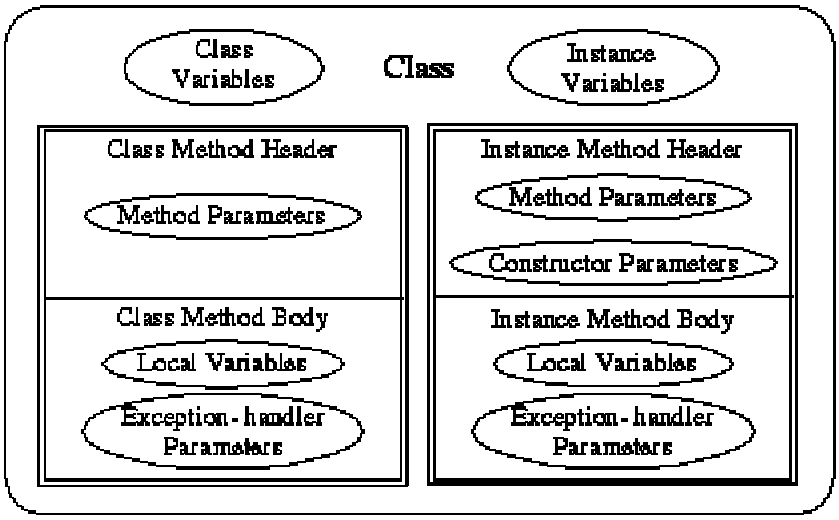

Figure 3-3: The Possible Location of the Six Types of Java Variables within a Java Class

Comments and documentation. Java programs can contain two kinds of comments: implementation and documentation comments (King et al. 1999). The documentation comments are used by Sun's Javadoc utility to generate class documentation in HTML format. The implementation comments are for human use only, and can be used to provide descriptions of files, methods, data structures, or algorithms; to describe business logic; or to mark code with programming notes. There are four styles

of implementation comments in Java: block, single-line, trailing, and end-of-line (King et al. 1999). All of these types of implantation comments, as well as documentation comments, can be found anywhere within a class.

Element Visibility

Access modifiers. There are three keywords in Java that specify the visibility of a Java class, interface, field or method (Gosling et al. 2000). These keywords are public, protected, and private. If none of these three keywords are used, the element is said to have "default" visibility, also called package or friendly visibility.

External visibility of classes. A top level class may have one of two visibilities: public or package, also called 'default' visibility. Public visibility is indicated with the keyword public and package visibility is not indicated (there is no keyword). Public classes are visible to all other classes, whereas classes with package visibility are visible only to classes within the same package. Because packages are outside the scope of this research, only classes with public visibility are used in the prototype application.

Internal visibility of class and instance variables. The values of all class and instance variables within a class can be read from and written to from any method within that class, no matter which access modifiers are used for the variables or methods. Table 3-1 summarizes this information and shows that for any of the possible 64 combinations of access to an internal class or instance variable from within a method in that class, the variable is accessed with read/write permission.

Table 3-1: Access granted to types of variables from types of methods when both reside within the same Java Class

Internal Method accessing	Internal Class Variables				Internal Instance Variables			
	Public	Protected	Private	Default	Public	Protected	Private	Default
Public class method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Protected class method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Private Class Method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Default Class Method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Public instance method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Protected instance method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Private instance method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access
Default instance method	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access	Read/ write access

External visibility of class and instance variables. Whether or not a field or method is visible from outside of a class depends on the access modifiers used on the fields and methods, and the inheritance and package relationships between the classes. All public fields and methods of class A are accessible from any method within class B, regardless of the relationships between the classes. All protected fields and methods of class A are accessible from any methods within class B if class B is either a subclass of class A, or class B is in the same package as class A. No private fields and methods of class A are accessible in any method of class B, regardless of the relationships between the two classes. All default fields and methods of class A are accessible to all methods of class B if class B is in the same package as class A, regardless of whether or not class B

is a subclass of class A. Table 3-2 summarizes this information and shows that there exists a degree of visibility among public, protected, private and default variables. Public class and instance variables are the most visible variables, followed by protected, default, and then private variables.

Table 3-2: Access granted to types of variables from methods when they reside in different Java Classes

External Method accessing Variable	Public Class or Instance Variable	Protected Class or Instance Variable	Private Class or Instance Variable	Default Class or Instance Variable
Any method in any non-subclass and in different package	Read/write access	No access	No access	No access
Any method in any external subclass in same package	Read/write access	Read/write access	No access	Read/write access
Any method in any subclass in different package	Read/write access	Read/write access	No access	No access
Any method in a non-subclass in same package	Read/write access	Read/write access	No access	Read/write access

Internal visibility of other types of variables. The other types of variables, method parameters, constructor parameters, exception-handler parameters, and local variables, are never accessible from outside of a class and do not use the access modifiers. These types of variables have fixed and limited scopes. Method parameters are accessible from anywhere within the same method's body. Constructor parameters are accessible from anywhere within the same constructor's body. Exception-handler parameters are accessible from anywhere within the entire catch statement body. Local variables are accessible from anywhere within the same block of code (between matching braces) in which it was declared. Table 3-3 summarizes the relative visibility of the Java variable types.

Table 3-3: Relative Visibility of all Java Variable Types

Variable Type (sorted from most visible to least visible)	Visibility external to the Class	Visibility internal to the Class
Public class or instance variables	Visible to all external classes	Visible anywhere within the class
Protected class or instance variables	Visible to all subclasses or same-package classes	Visible anywhere within the class
Default class or instance variables	Visible to all same-package classes	Visible anywhere within the class
Private class or instance variables	No visibility	Visible anywhere within the class
Method or constructor parameters	No visibility	Visible anywhere within the method/constructor body
Exception-handler parameters	No visibility	Visible anywhere within the catch statement body
Local variables	No visibility	Visible anywhere within the same block

Class and instance methods. All class and instance methods can be called by any other method within the same class. The external visibility of class and instance methods depend on the access modifiers used with them. Public methods in a class A can be called by any method in class B regardless of the relationship between them. Protected methods in class A can be called by any method in class B if class B is a subclass of class A, or if they are in the same package. Default methods of class A can be called by any method in class B if the classes are in the same package. No private method in class A can be called from within class B. This information is summarized in Table 3-4. As was the case with class and instance variables, there is a degree of visibility among the types of class and instance methods. Public methods are the most visible from outside the class, followed by protected, default and private methods.

Table 3-4: Access Granted to Types of Methods

Method accessing Method	Public Class or Instance Method	Protected Class or Instance Method	Private Class or Instance Method	Default Class or Instance Method
From any method within the same class	Visible	Visible	Visible	Visible
From any method in any non-subclass and in different package	Visible	No visibility	No visibility	No visibility
From any method in any external subclass in same package	Visible	Visible	No visibility	Visible
From any method in any subclass in different package	Visible	Visible	No visibility	No visibility
From any method in a non-subclass in same package	Visible	Visible	No visibility	Visible

CHAPTER 4
JAVA PROGRAMMING GUIDELINES

Benefits of Programming Guidelines

Coding guidelines exist to solve some of the problems resulting from the way in which software is typically developed and maintained. Many programmers work in teams on large software projects. The implication of this is that these programmers need to be able to understand each other's code writing styles quickly in order to be more productive. Another software development reality is that often the programmers maintaining software are not the same programmers who developed it (King et al. 1999, Sommerville 2001). The maintainers need to be able to understand the developers' code. Software maintenance is an important consideration because it may account for 65-75% (Sommerville 2001), or as much as 80% (King et al. 1999), of the total lifetime cost of the software. In addition, sometimes the source code is the documentation either because separate documentation was never written, is out-of-date or is lost (Sommerville 2001). The implication of this is that the software source code should be self-documenting.

The potential benefits of guidelines include code that is consistent, takes less time to understand, permits a deeper understanding, is easier to maintain, and is less expensive overall (Ambler 2000, Fussell 1998, King et al. 1999). In addition, having to conform to these coding guidelines can increase the developer's application, algorithm and object design and programming skills (Fussell 1998). King et al. (King et al. 1999) point out that adherence to code conventions reflects an organization's degree of professionalism.

General Object-Oriented Programming Guidelines

A review of the literature concerning general object-oriented programming guidelines reveals a small common set of guidelines. These guidelines include: decrease extraneous coupling among classes, increase class cohesion, minimize the public interface, hide and centralize any variable information, minimize the complexity of code and objects, code in a consistent style, and provide adequate documentation (Chidamber and Kemerer 1994, Henderson-Sellers 1996, Rosenberg 1998, Systä et al. 2000).

Eliminate Extraneous Class Coupling

Coupling refers to the degree of interdependence or strength of association between classes or objects. When a method in one class makes calls to a method in another class, or accesses attributes of the other class, the two classes are coupled. Two objects are coupled when they send messages to each other. High degrees of coupling between classes can lead to less understandable code, more testing required, less reusable code, less modular code, and less maintainable code (Venners 1998). Effect of change or error in one class may propagate to a large number of other classes if they are highly coupled.

The degree of interdependence between classes should be minimized to a certain degree. There is a certain amount of coupling that is necessary so that the software is useful (Henderson-Sellers 1996). Some authors differentiate between the necessary or "inheritance" coupling, and the unnecessary, or "noninheritance" coupling. Classes that are related through an inheritance hierarchy, are of necessity highly related to each other.

Maximize Internal Class Cohesion

The 'cohesion' of a class has two different meanings in the guideline literature. The first meaning refers to the degree in which methods and fields within a class are related to each other. The second meaning refers to the number of major functions performed by the class. Both meanings can be subsumed into a central idea in object oriented design that good object design requires following a procedure of first identifying the major responsibilities, dividing the responsibilities among objects, and lastly identifying the internal data needed by the object in order to perform the services it is responsible for. A high degree of cohesion is an indicator of good object or class design.

Cohesion, defined by the degree of relatedness among a class' methods and variables relates to the conventional notion of consistency found in software engineering, and also to the notion of encapsulation-keeping data and the methods that act on the data together in one place (Rosenberg 1998). Cohesion, defined by the number of major functions performed by a class, is an indicator of how hard the class will be to understand. Classes that have more than one main function are not cohesive and should have been split into multiple classes. A highly cohesive class is easier to understand and maintain.

Minimize Public Interface

The number of public class methods nd variables should be minimized for several reasons (Lorenz and Kidd 1994, Rosenberg 1998, Daconta et al. 2000, Ambler 2000). From the perspective of a developer trying to understand the code, a smaller interface means that the class is easier to understand. From the perspective of someone maintaining the code, fewer public methods and variables means that it is easier to change and debug.

Minimizing the public interface is related to the guideline on minimizing class coupling, as the interface is the "point of coupling" (Venners 2002).

Hide Information

A key concept in object-oriented programming is information hiding. Any data that is likely to change should be separate and hidden from the public interface (Neal et al. 1997a, Rosenberg 1998, Haahr 1999). The internal data structure and implementation of the public interface should not be revealed in the interface, so that these internals can be changed without affecting applications using the interface, so that the interface is simpler to understand, and so that reuse of the class takes less effort. This is probably the least-disputed object-oriented programming guideline.

Minimize Complexity

Minimization of program and algorithm complexity was a guideline for functional programs before it was a guideline for object-oriented programs. The unique characteristics of object-oriented programming add to this traditional guideline the recommendation that the complexity of objects should be minimized as well. Complexity is an abstract concept and can mean different things to different authors, but it generally means the time to understand, debug and test a program or parts of it.

Code in a Consistent Style

There are many functional and object-oriented programming guidelines based on programming style, such as naming, documentation and code layout guidelines. While it would be too restrictive to require all programmers to follow the same style guidelines, what is important is that there exist style guidelines for all software development organizations and large software projects, and that these guidelines are consistently

followed (Haahr 1999). The potential benefit of consistent coding style is less time to understand and debug a program.

Provide Adequate Documentation

Many object-oriented authors contend that a high percentage of comments throughout a program can make the program easier to understand (Ambler 2000, Rosenberg 1998). A less popular argument is that if the code were written well it would not need very much documentation (King et al. 1999, Feigenbaum 2000). The discrepancy may be related to a tendency in the guidelines literature not to differentiate between documentation that describes the background or business logic behind code (which is generally seen as a good thing) versus documentation that explains hard-to-read code (which is generally seen as an indicator of bad code-writing). Nevertheless, the potential benefits of highly documented code include code that is easier to understand how it works and some of the reasons why it was programmed the way it was. In addition, programmers can use comments to mark code that may change in the future.

Java-Specific Programming Guidelines

There are numerous guidelines for Java programming. Of necessity, this research focuses on a small subset of them. Guidelines for the following Java elements are considered outside of the research scope: method parameters, interfaces, packages, compilation units, test harnesses, loop statements, conditional statements, exceptions, threads, abstract classes, inner classes and inheritance design. Guidelines for the following Java elements are considered within the scope of this research because they were identified as the main elements within a Java class in chapter three, and beginning Java programmers will need to fully understand these elements: classes, class variables,

instance variables, class methods, and instance methods. Local variables are also included in the scope of this research so as to cover all sub-class variable scopes in the prototype application.

Guidelines for Classes

Class naming. Java classes should have natural descriptive English names that are nouns or noun phrases (Badeaux 1999, Gosling et al. 2000). The first letter should be capitalized, and the rest of the name should be in mixed case (Eckel 1998, Lea 2000, Gosling et al. 2000, Ambler 2000). The names should not be too long - 20 characters or less (Loeffler 2000). Acronyms and abbreviations should be avoided unless the abbreviated version is more widely used than the long version, such as HTML or URL (King et al. 1999).

Ordering of methods and variables within the class. There are many guidelines on ordering methods and variables within a class, but there is no consensus as to the ordering. There are some common features among the majority of guidelines on this subject. All the class and instance variables should be grouped together into one location, either above or below all the methods. Methods should be grouped into categories, either by their visibility, by the function they serve, or alphabetically. Elements that a reader of the file would be most interested in, such as the constructors and destructors, should be near the top of the file. Five suggestions for the ordering are shown in Table 4-1. What is evident from Table 4-1 is that while there is no single ordering guideline, the methods and variables should be grouped into categories and be used consistently.

Table 4-1: Five variations on a guideline for the ordering of methods and variables within a class

Guideline Source	Ordering Guideline
King et al. 1999	1. public class variables 2. protected class variables 3. default class variables 4. public instance variables 5. protected instance variables 6. default instance variables 7. instance instance variables 8. constructors 9. methods grouped by functionality, not be accessibility
Ambler 2000	1. constructors 2. finalize() 3. public methods 4. protected methods 5. private methods 6. private fields
Ambler 2000 (alternate guideline)	1. constructors 2. finalize() 3. class methods in alphabetical order 4. instance methods in alphabetical order 5. private fields
Badeaux 1999	1. constructor/destructor methods 2. factory methods (usually class methods) 3. accessor methods 4. standard methods 5. debugging methods
Loeffler 2000	1. fields 2. public constructor(s) 3. finalizers (optional) 4. public methods. 5. accessors and mutators 6. other methods 7. a main method (optional)

Size of class. The size of a class should not be overlarge, so that it is easier for developers and maintainers to understand it (Rosenberg 1998). Class size can be measured in several ways: by the number of lines of code, by the number of methods, or by the number of variables.

Lines of code may refer to all lines within the source code file, only the non-comment lines, only non-blank lines, only executable statements, or a combination of these. King et al. (King et al. 1999) recommends that a source code file be no more than

2000 lines of code, but does not specify what is meant by 'lines of code'.

Several authors state that the number of methods in a class is a good predictor for how much time and effort will be required to understand the class (Henderson-Sellers 1996, Rosenberg 1998). The larger the number of methods, the more impact there will be on any subclasses of the class (Rosenberg 1998). A class with many methods is probably application-specific, and therefore less likely to be reusable (Lorenz and Kidd 1994, Rosenberg 1998). A large number of methods can mean that the class should have been designed as multiple classes, although too few methods may mean that the class should have been combined with other classes (Henderson-Sellers 1996). The number of methods can be counted either in absolute numbers or as a percentage of another sub-class element (Lorenz and Kidd 1994).

The number of instance variables in a class can also be used as an indicator of class size. Too many instance variables can mean that the programmer is focusing on the data instead of the services the class performs (Lorenz and Kidd 1994). Classes with many instance variables can be harder to reuse (Lorenz and Kidd 1994).

While qualitative guidelines for class size are numerous, quantitative are harder to find. Table 4-2 shows the results of an experiment by (Elish 2001) that shows descriptive statistics for 100 randomly-chosen open-source Java files. These numbers can be used to approximate thresholds for acceptable class size. Interestingly, (Elish 2001) found a positive correlation between the tendency to not follow good coding guidelines and lines of code, number of methods, and number of variables in a class.

Table 4-3 shows recommended thresholds for acceptable class size for C++ and Smalltalk by (Lorenz and Kidd 1994). It is expected that these numbers can also aid in

developing thresholds for Java. Of note is the recommendation that there be few class methods as compared to instance methods. This guideline is in agreement with those of many authors for Java.

Comparing Tables 4-2 and 4-3 it is evident that in practice Java programmers tend to have fewer methods and many more variables than recommended by Lorenz and Kidd (Lorenz and Kidd 1994). Unfortunately Elish (Elish 2001) does not indicate whether local variables were counted in the number of variables, so the inclusion of local variables may account for the large number of variables in the median Java class.

Table 4-2: Descriptive statistics for 100 Java classes on class size

Statistic	Lines of code	Number of methods	Number of variables
mean	269.80	10.00	20.90
standard deviation	323.20	7.10	18.60
mode	107.00	8.00	3.00
minimum	18.00	1.00	0.00
1st quartile	108.00	6.00	8.00
median	179.00	8.00	18.00
3rd quartile	280.50	12.00	25.50
maximum	1880.00	50.00	109.00

Source: Elish 2001.

Table 4.3: Thresholds for measurements of class size for C++ and Smalltalk programs

Measurement	Threshold
Number of public instance methods (C++ only)	No more than 20
Number of all instance methods	No more than 20
Average number of public instance methods within a group of classes (C++ only)	No more than 12
Number of instance variables	No more than 3
Average number of instance variables in a group of classes	No more than 3
Number of class methods (absolute numbers)	No more than 4
Number of class methods (relative to the number of instance methods)	No more than 20%
Number of class variables	No more than 3
Average number of class variables in a group of classes	No more than .1

Source: Lorenz and Kidd 1994.

Class Documentation. Documentation comments (those starting with /** and ending with */) are those used by the Javadoc utility from Sun to generate external class documentation. Documentation comments should appear before the class declaration (Lea 2000, Ambler 2000), as well as before each method declaration (Ambler 2000), or at least before all public methods (Badeaux 1999). Elements to include in the class documentation include the purpose of the class, known bugs, the development/maintenance history, guaranteed invariants, usage instructions, and the concurrency strategy if applicable (Ambler 2000, Lea 2000).

Guidelines for Class and Instance Variables

Naming class and instance variables. The names of class and instance variables should be descriptive English names (Ambler 2000, Gosling et al. 2000, Loeffler 2000). Some authors recommend a leading or trailing underscore for instance variables (Ambler 2000), while others explicitly recommend against this (King et al. 1999). The first letter should be lowercase and the rest of the name should be mixed case (Eckel 1998, King et al. 1999, Loeffler 2000, GSS 2001). Variables that are collections, such as arrays and vectors should have plural names (Ambler 2000, Loeffler 2000). When declaring an array, use the form: `Type[] arrayName` instead of: `Type arrayName[]` [Lea 2000]. Constants should be in all uppercase with underscores between words (King et al. 1999, Ambler 2000, Feigenbaum 2000, Gosling et al. 2000, Lea 2000, Loeffler 2000, GSS 2001). Name hiding is discouraged (Ambler 2000, Feigenbaum 2000). This includes using a variable name that was used in a superclass.

Declaration and initialization of class and instance variables. One variable declaration per line is recommended because it encourages the use of end of line

comments explaining the variable (King et al. 1999, Feigenbaum 2000). Techniques to especially avoid include declaring variables of different types on the same line, and assigning the same value to multiple variables on the same line because they are hard to read (King et al. 1999).

Attention should be paid to ensure that class variables have valid values because class variables may be accessed before instances of a class have been created (Ambler 2000, Lea 2000). Lea (Lea 2000) suggests using 'static initializers' to initialize class variables which would run when a class is loaded. This is only of concern if the class variables are visible.

Visibility of class and instance variables. Many Java guideline authors state that all class and instance variables should be declared private (Ambler 2000, Badeaux 1999, Tyma et al. 1996). Others recommend that they at least not be declared public (Fussell 1998, Lea 2000). The reasons for restricting access to these variables include keeping control over the internal class structure, ensuring that variables always have valid values, and promoting class reusability and maintainability (Lea 2000).

Rather than allowing external classes to access the variables directly, accessor and mutator methods (getters and setters respectively) are recommended. Strict adherence to only allowing access to these variables through methods can simplify the implementation of event notification when an attribute changes (an important Java Beans concept) and lazy initialization of attributes (Loeffler 2000). There need not be accessor and mutator methods for every class and instance variable, however. It is a better practice to only create an accessor or mutator method on a need-access basis (Feigenbaum 2000).

Some guideline authors point out situations when these variables should not be private. Unit testing is easier with variables of default visibility (Loeffler 2000). When a class is essentially a data structure with no behavior it is appropriate to make the instance variables public (King et al. 1999). Private variables are of no use to subclasses of the class, so protected may be better to use in cases where inherited variables are desired (Lea 2000). When performance is an issue, allow internal and external methods to access variables directly (Ambler 2000).

Some guideline authors caution against the misuse of public class variables (Venners 1999, Lea 2000). Unless they are also declared as 'final', they act as global variables, which are the antithesis of data hiding. Public final class variables are essentially constants and their use is encouraged, as opposed to the less flexible practice of hard-coding literals that may change (King et al. 1999). Feigenbaum (Feigenbaum 2000) points out that it is a better practice to make constants private and only allow access through accessor methods. This way they can be made non-constant or a different internal data type in the future without causing many other code changes.

Guidelines for Local Variables

Naming local variables. Most Java guidelines authors agree that in general local variables should be named similarly to class and instance variables, using descriptive English names with the first letter lowercase, and the rest of the name in title case (Ambler 2000, Loeffler 2000). This naming guideline is relaxed for specific types of local variables in which the names can be very short or single characters. A convention for loop counters is to use the names 'i', 'j', or 'k' (Ambler 2000, Gosling et al. 2000, Loeffler 2000). Streams are typically named either 'in', 'out', 'inOut', 'inputStream',

'outputStream', or 'ioStream' (Ambler 2000, Gosling et al. 2000). Exception objects are usually named 'e' (Ambler 2000, Gosling et al. 2000). Gosling et al. (Gosling et al. 2000) recommend using single character names for many specific local variables, such as 'f' for a float, while the majority of Java guidelines authors recommend using longer descriptive names (Ambler 2000, King et al. 1999).

Declaring and initializing local variables. Only one local variable should be declared per line so as to encourage end of line documentation describing the variable (King et al. 1999, Ambler 2000). Practices that should be especially avoided include declaring variables of different types on the same line (King et al. 1999), reusing a variable name for a different function (Ambler 2000, Lea 2000), and reusing a variable name that exists in an outer scope (King et al. 1999). Feigenbaum (Feigenbaum 2000) points out that there are no advantages in Java to reusing variable names due to its stack-frame nature, and that creating more specific and self-describing variable names, rather than reusing generic names, makes the code easier to understand.

There are different camps in the guideline literature as to where a local variable should be declared. Some believe that all local variables (except the initialization variable of a for loop) should be declared at the beginning of blocks (King et al. 1999, Loeffler 2000). The reasoning behind this is make the code more portable by having all the variable declarations in one place in each block. Others recommend that they be declared immediately before their use (Ambler 2000). Still others recommend that they be declared only at the point in code where the initial value is known so that they can be declared and initialized on the same line (Fussell 1998, Lea 2000). The reason for wanting to initialize local variables as soon as they are declared is that Java does not

guarantee default initializations of local variables (Loeffler 2000). Because of this, the authors agree that a local variable should be initialized on the same line that they are declared, no matter where they are declared.

Guidelines for Class and Instance Methods

Naming class and instance methods. The names for class and instance methods should be similar to those of class and instance variables: descriptive English names, with the first letter lowercase, and the rest of the name in title case (Eckel 1998, Badeaux 1999, Ambler 2000, Loeffler 2000). The difference between the method and variable names is that method names should be strong active verbs or predicate phrases (King et al. 1999, Ambler 2000, Feigenbaum 2000, Gosling et al. 2000, GSS 2001). Method names do not need to be verbs if there are long-held naming conventions that can be used instead, such as the `sin()` and `cos()` methods of the class java.lang (Gosling et al. 2000).

In addition to that general naming guideline, there are guidelines for naming specific types of methods. Accessor instance methods should be prefixed with 'get' (Badeaux 1999, Gosling et al. 2000, Loeffler 2000, GSS 2001). Mutator instance methods should be prefixed with 'set' (Badeaux 1999, Ambler 2000, Gosling et al. 2000, Loeffler 2000, GSS 2001). Boolean instance methods should be prefixed with 'is', 'can', or 'has', although 'is' is preferred because of restrictions with the BDK (Bean Development Kit) (Ambler 2000, Gosling et al. 2000, Loeffler 2000). Convertor methods that convert an object to a particular format should be prefixed with 'to' (Gosling et al. 2000, Lea 2000). Methods that return the length of something should be named 'length' (Gosling et al. 2000).

Visibility of class and instance methods. The visibility of class and instance methods should be as restrictive as possible (Ambler 2000). All utility methods should be made private (Tyma et al. 1996). Use protected visibility for all methods that must be visible only within the package or subclasses of the class, and protected visibility only when the method needs to be visible to external (non-subclass, non-same-package) classes (Ambler 2000).

Size and complexity of class and instance methods. A method should be fully understandable by another programmer within thirty seconds (Ambler 2000, Loeffler 2000). Longer methods are more difficult to understand (Fussell 1998, Rosenberg 1998) and may indicate that function-oriented code is being written (Lorenz and Kidd 1994). Recommendations for the maximum length of a method include that it should be less than a screen long (Ambler 2000), no more than 50 non-commented lines (Feigenbaum 2000), no more than 25 lines (Tyma et al. 1996), no more than five or ten lines (Haahr 1999), or until it becomes too difficult to follow (Tyma et al. 1996). Elish (Elish 2001) found the median number of lines in a method to be 22.375. An alternative way to describe method size is the number of statements as a percentage of the number of message sends, which should be about 80% (Lorenz and Kidd 1994).

The width of a line of code within a method should be visible on the screen without having to scroll to view it (Ambler 2000). Line width should not exceed 80 characters (Badeaux 1999, Haahr 1999, King et al. 1999, GSS 2001), or at most 100 characters (Feigenbaum 2000). Each line should perform only one function (Ambler 2000), contain only one statement at most (King et al. 1999), and not contain too many operations (Badeaux 1999).

Cyclomatic complexity, which counts the number of decision points, is traditionally used to measure the complexity of a function or method (Lorenz and Kidd 1994). Lower cyclomatic complexity implies decreased testing and increased understandability (Rosenberg 1998). Lorenz and Kidd (Lorenz and Kidd 1994) suggest that the number and type of message sends is a better indicator of method complexity for object-oriented languages than cyclomatic complexity.

Methods should only perform one function. For example, the two Stack class methods, `top()` and `removeTop()`, are better than the method `pop()` which performs both tasks (Lea 2000). Methods should contain only one entry and one exit point to make debugging and testing easier and to make it more understandable (Feigenbaum 2000).

Indenting and whitespace within class and instance methods. Top-level classes and interfaces are not indented. All other members should be consistently indented units of two to four spaces representing the scope of a code block (King et al. 1999). Use of smaller numbers of spaces reduces the chance of having to split lines, but Sun recommends four spaces (GSS 2001). Special characters like tabs or page breaks should be avoided because they tend to be system-specific (GSS 2001).

Add a space around binary operators, after commas, after keywords and before opening parenthesis (for example: `while (true)`), between expressions in a for statement, after casts, after colons, and after semicolons (King et al. 1999, GSS 2001). Do not add spaces between a method name and the parenthesis, between unary operators and their operands, and between the binary operator '.' and its operands (King et al. 1999, Feigenbaum 2000).

Logically related groups of statements within a method should be separated from each other by a blank line (Loeffler 2000). All control structures should be separated from the rest of the method by a blank line (Ambler 2000, Loeffler 2000). The methods themselves should also be separated from each other by one to two blank lines (King et al. 1999).

Documenting class and instance methods. Every method, or at a minimum all public methods, should be preceded by a javadoc comment header (Badeaux 1999, Ambler 2000, Lea 2000). Within the method internal documentation (C-style and end-of-line) should be used to document all control structures, local variables, and difficult or complex code (Ambler 2000). Table 4-4 shows descriptive statistics on the comment density of 100 randomly-chosen Java open-source programs. Elish (Elish 2001) found no correlation between the comment density of a class and the number of violations found in that class.

Table 4-2: Descriptive statistics for 100 Java classes on comment density

Statistic	Comment Density
mean	15.2%
standard deviation	12.2%
mode	0.0%
minimum	0.0%
1st quartile	6.9%
median	12.3%
3rd quartile	21.3%
maximum	72.4%

Source: Elish 2001.

CHAPTER 5
PROTOTYPE APPLICATION

Goals of the Prototype Application

There are three main goals of the visualization application. The first goal is to explore new ways to represent the *logical structure* of a Java class. The second goal is to explore ways to represent the *quality* of a Java class' design and coding. The third goal is to explore ways to to represent *prioritized improvement recommendations* for a Java class' design and coding. The tools used for this representation are the visualization presentation techniques discussed in chapter two: three spatial dimensions, color, visual metaphors, visual clarity, visual codings and good layout principles.

For the purpose of this research the 'quality' of a Java class' design and coding is defined as the degree of adherence to selected object-oriented and Java guidelines, which were discussed in chapter four. The fact that many of these guidelines are subjective and seemingly arbitrary is not of central importance to this research. The ability of humans to delete or change the parameters of guidelines is included in the prototype application, as well is the ability to modify the relative importance of each guideline.

Selected Object-Oriented and Java Guidelines

The application's programming guidelines were chosen from the guidelines discussed in chapter four. They were selected so that every possible category listed in chapter four under the heading 'general object-oriented guidelines' was covered by at least one selected guideline, and so that every major Java class element discussed in

chapter three was covered. The guideline 'eliminate extraneous class coupling' was not covered because of this research's scope limitation to the class level.

Table 5-1 lists the seven selected guidelines, how compliance with the guideline is measured, and the applicable Java class element(s). The Java class elements listed in the table are the elements that are being evaluated by the guideline, which are not necessarily the elements used in evaluating the guideline. For example, the first guideline listed in Table 5-1, 'maximize class cohesion' is an indicator of how well the class was designed. It requires the class' methods and variables to compute the degree of guideline compliance, but it is the class and not the methods and variables that is the applicable Java element.

Table 5-2 shows the number of guidelines selected for each of the Java class elements. The table illustrates that the number of guidelines per Java class element roughly corresponds to the importance of the element within the class. For example, local variables have the least number of selected guidelines, while classes have the most.

Table 5-1: Selected Design and Coding Guidelines for Application

Guideline	Compliance Metric	Element Evaluated
1. Maximize class cohesion	1. Number of disjoint sets of class or instance variables and methods per class (all methods and variables that call each other are in the same set)	class
2. Minimize size of public interface	2. Number of public class or instance methods and variables per class	class
3. Hide information	3a. Ratio of class or instance variables that are public, protected and/or default to all class or instance variables	class
	3b. Number of global variables per class (public class variables that are not declared final)	class
4. Minimize complexity	4a. Number of lines of code per class (counting blank lines and comments)	class
	4b. Number of lines of code per method (counting blank lines and comments within the method scope)	class methods and instance methods

Table 5-1-continued.

Guideline	Compliance Metric	Element Evaluated
	4c. Maximum line length per method (counting leading white space and trailing comments)	class methods and instance methods
	4d. Average line length per method (counting leading white space and trailing comments)	class methods and instance methods
5. Code in a consistent style	5a. Number of method naming policies (for example all title case, or all having the first letter capitalized)	class
	5b. Number of variable naming policies (for example all title case, or all having the first letter capitalized)	class
6. Provide adequate documentation	6a. Comment density per class (Ratio of lines of code that are comments or have end-of-line comments to all lines)	class
	6b. Comment density per method (Ratio of lines of code that are comments or have end-of-line comments to all lines)	class methods and instance methods
	6c. Existence of a documentation heading for the class (regardless of whether they contain javadoc comments)	class
	6d. Existence of documentation headings for each class and instance method (regardless of whether they contain javadoc comments)	class methods and instance methods
	6e. Existence of end-of-line comments for each class and instance variable	class variables and instance variables
7. Follow naming conventions	7a. Class name compliance (first letter uppercase, rest of name title case)	class
	7b. Class and instance method name compliance (first letter lowercase, rest of name title case)	class methods and instance methods
	7c. Class and instance variable name compliance (first letter lowercase, rest of name title case) (Exception are constants which should be uppercase)	class variables and instance variables

Table 5-2: Number of selected design and coding guidelines for each Java class element

Java Class Element	Number of Guidelines	Guidelines
class	10	1, 2, 3a, 3b, 4a, 5a, 5b, 6a, 6c, 7a
class methods	6	4b, 4c, 4d, 6b, 6d, 7b
instance methods	6	4b, 4c, 4d, 6b, 6d, 7b
class variables	2	6e, 7c
instance variables	2	6e, 7c
local variables	0	-

Selected Java Classes

Thirteen Java programs were selected for input into the visualization application. The goal behind the class selection process was to have programs with a mix of compliance with the selected Java programming guidelines. Characteristics of a class that disqualified it from selection were it being an abstract class, it being an interface, it having inner classes or interfaces, or it having less than 2 methods. The number of methods requirement is meant to filter out any programs that are not performing a useful function, such as the 'Hello World' programs.

Six of the classes were chosen with the expectation that they would not comply with very many of the guidelines. They were created in various computer science classes as opposed to being intended for public release. In addition, these five classes were each designed and developed by multiple developers.

Another seven programs were selected with the expectation that they would meet some if not all of the selected guidelines. Five of these programs are part of Java tutorials, the other two are part of publicly available applications.

Table 5-3 lists the selected Java input programs, their authors and creation purpose. Table 5-4 shows descriptive statistics for the input programs.

Table 5-3: The selected Java programs for use as input to the visualization application

Program name (Application if applicable)	Author(s)	Lines of Code	Program Purpose
BigCube.java	Sun Microsystems, Inc.	126	Downloadable Java 3D Demo
Grammar.java	UF computer science graduate students	50	UF computer science graduate course project
HazardChecker.java	UF computer science graduate students	618	UF computer science graduate course project
Hazard.java	UF computer science graduate students	91	UF computer science graduate course project
HtmlLink.java (part of Jigsaw 2.2.0)	W3C (World Wide Web Consortium)	140	Public release
IWonEvent.java	Sun Microsystems, Inc.	32	On-line Java tutorial
NapsterSong.java (part of jNapster 0.1)	Harikrishnan Varma, developer	324	Public release
NotaryPublic.java	Sun Microsystems, Inc.	59	On-line Java Tutorial
Scanner.java	UF computer science graduate students	375	UF computer science graduate course project
ScoreBoardSimulator.java	UF computer science graduate students	592	UF computer science graduate course project
StripQualifiers.java	Bruce Eckel, author of Java programming books	49	On-line Java tutorial
TinyParser.java	UF computer science graduate students	1430	UF computer science graduate course project
website.java	Java Developer's Resource	46	On-line Java tutorial

Note: Lines of code includes blank and commented lines

Table 5-4: Statistics on the number of lines of code for the selected Java input programs

Statistic	Value
Mean	302.31
Median	126
Minimum	30
Maximum	1430
Range	1400
Standard Deviation	396.8

Note: Number of lines of code includes blank and/or commented lines

Development of Techniques for Presentation of Program Analysis

The visualization focuses on showing three things about a Java program: the visibility of its members, the message connections within the class, and the degree of compliance with the selected Java programming guidelines. The presentation techniques

used to show these things are use of color, visual metaphor, filtering, size, shape, position, line width, and use of three dimensions.

The visual metaphor of a traffic light and a color scale are combined to represent the degree of compliance with the selected guidelines. The familiarity of the traffic light's unfavorable red, warning yellow and favorable green are adapted to lessen the cognitive load needed to understand the visualization. For the visualization, green represents a high degree of guideline compliance, yellow represents a medium degree of guideline compliance, and red represents a low degree of guideline compliance. A color scale between these three colors was created to represent the semantic distance between degrees of compliance. Figure 5-1 shows the color scale used for the visualization.

Figure 5-1: Color scale used for visualization
Notes: There are 25 colors in the scale. The far left color, dark green, represents total compliance with the guidelines. The far right color, dark red, represents total non-compliance with the guidelines. All in-between colors represent a strictly decreasing degree of compliance with the guidelines.

The visual metaphor of a boundary, wall, or fence, in combination with size, represents the semantic distance between a Java element that is relatively compliant with programming guidelines, and one that is not. The idea is that a visual fence can be used as a quick reference to identify the non-compliant Java elements. If the element is so large that it 'crosses' the fence, it is a non-compliant element. Figure 5-2 illustrates how size and the fence show degree of guideline compliance.

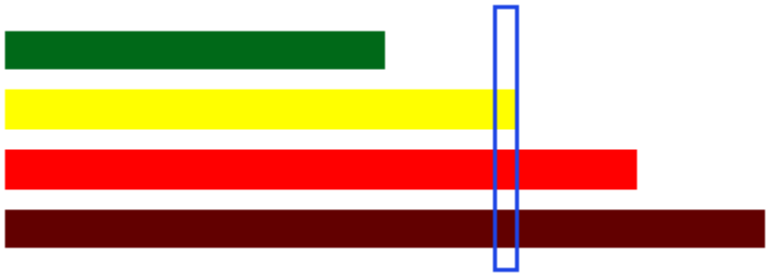

Figure 5-2: Size and fence metaphor used for visualization
Notes: The blue rectangle in this figure represents the fence. The green rectangle represents a 100% guidance-compliant element. The dark red rectangle represent a 0% guidance-compliant element. The yellow rectangle represents an element that is bordering on non-compliance, and the red rectangle is a non-compliant element.

Shapes are used to distinguish between Java elements. A class or instance variable is depicted as a cylinder. A class or instance method is depicted as a hexahedron, as is the class itself. The class and its methods can be distinguished from each other in the visualization because the class' methods are shown inside of the class, as are its variables.

Position is used in the visualization to show both ordering and the degree of relatedness. Class and instance variables are ordered by the number of times they are called in the class. Class and instance methods are ordered by the sum of the number of times they call another variable or method in the class, and the number of times the method itself is called by another method in the class. The proximity of the methods and the variables to each other represents their degree of relatedness or coupling to each other. The idea is that the methods and the variables furthest away from each other on the "outskirts" are used less within the class. Figure 5-3 illustrates this concept.

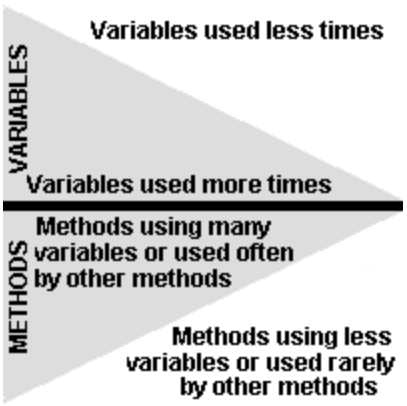

Figure 5-3: Position and degree of relatedness as used in the visualization
Notes: The black horizontal line in this figure is used to separate the physical area for the class and instance variables above the line, and the class and instance methods below the line. The gray shaded area in the center shows that the variables and methods that are used the most often or make the most calls are located near each other.

Filtering is used to show the visibility of the Java element from one of four different "locations" - within the class, from a subclass, from a class in the same package, or from an unrelated class. By "visible" it is meant that the method or variable is in the same scope. The filtering is conducted by using a "coating"of black color over the member's label id to represent the fact that the member is not visible. Figure 5-4 illustrates this concept.

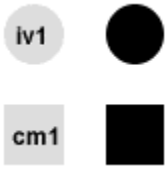

Figure 5-4: Filtering in the visualization
Notes: The top left circle represents an instance variable that is currently visible. The top right circle represents a variable that is not visible. The bottom left square represents a visible class method. The bottom right square represents a method that is not visible.

Line width is used to represent the number of message connections between the variables and methods. The thicker the line, the more messages the single line represents.

Rating of Java Class Elements

A grade is used to rate each class or instance variable and method, as well as the class itself. The minimum grade of a Java element is 0, the maximum is 100. The grade is based on the element's guideline compliance. The default penalties for breaking guidelines are not uniform because of several reasons. The acceptance level of the guidelines is not uniform. Some Java guidelines, such as naming conventions, are almost universally accepted. Other guidelines such as comment density are less accepted and therefore should have less harsh penalties.

Because the guidelines and the penalties for breaking guidelines have a subjective nature, they are customizable in the application. There are four configuration files in the application. The file, "defaultMetrics", contains default parameters for the guidelines, such as the maximum number of public members in the class. Another customizable file, "customMetrics" overrides the parameters set in "defaultMetrics". The file, "defaultGrading", contains the default penalties for breaking each guideline, such as -5 points for breaking the guideline on the maximum number of public members in the class. Another customizable file, "customGrading" overrides the default penalties. All the default settings are based on the research conducted for this paper. The contents of the configuration files are shown in Appendix B.

Development of Application

Program Description

The application consists of seventeen Java programs. Additional files include a driver ("test"), a makefile, a makefile utility file (make.cfg"), the thirteen input programs and four configuration files, which were discussed in the last section. The requirements

for running this application include a Java 2 jre, and the Java 3D library, as this was the 3D graphics library (on top of OpenGL) used for the visualization. Table 5-5 describes the purpose of each Java program.

Table 5-5: Description of the Application's Java programs

Java Class	Purpose
JCallMatrix	Used to record all the message calls between the variables and methods within a class
JClass	Contains all the class-level information about a class
JClassMethod	Contains information about a single class method
JClassVariable	Contains information about a single class variable
JColorConstants	Contains color information used in the visualization
JCritique	Critiques a Java element using the configuration files
JGrammar	Contains information about the grammar of the Java language
JInstanceMethod	Contains information about a single instance method
JInstanceVariable	Contains information about a single instance variable
JListener	Used by the GUI to change the visualization based on the viewing location pull-down menu
JMethod	Contains information about a single method
JModeler	Draws the visualization and creates the GUI
JParser	Parses the input Java program and creates the JClass object from it
JParseState	Contains the current state (scope, location) within the parsing of the input Java program
JStatistics	Runs statistics on the JClass object based on needs of the configuration files
JVariable	Contains information about a single variable
JVisualizer	The main program

Program Flow

When the program is started, it parses the file and builds a data structure (a JClass object) of all the necessary information about the class and its members. It then constructs a matrix of all the calls between the class or instance methods and variables. Then statistics are run on the JClass object so that compliance with guidelines can be measured. Next, each member of the class, and the class itself are given a grade, and the problems with each of these elements are recorded. Lastly, the application GUI is drawn,

together with the visualization and a report on the class being visualized. Figure 5.5 shows this sequence of events.

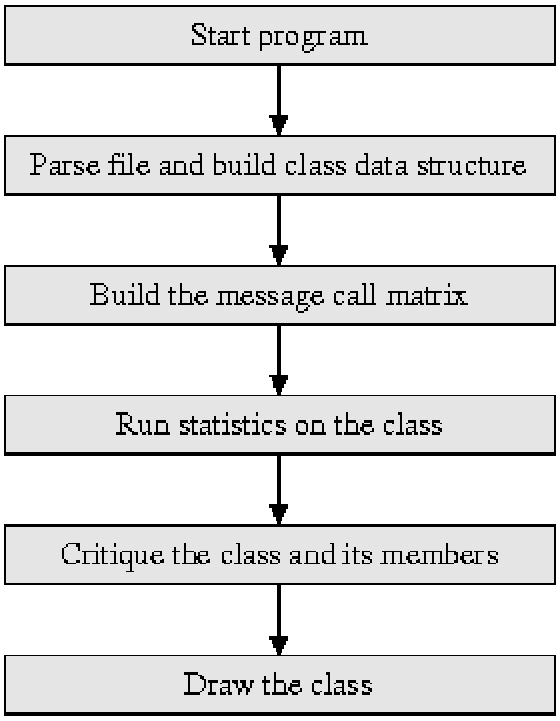

Figure 5.5: Program Flowchart for Application

GUI Layout

The application's GUI is composed of six sections as shown in Figure 5-6. The top section contains a pull-down menu that is used to change the viewing location. The viewing location affects what is visible in the visualization. The left-center section is the 3D visualization. The right-center section contains a report on the class being visualized. The report starts by listing the members within the class that received the lowest grades, and then follows by a summary of statistics and criticism for the class and each member. The left-bottom section contains instructions on how to interact with the visualization. The supported interactions include rotation, transpose and zooming. The bottom-right section is a legend to the colors used in the visualization.

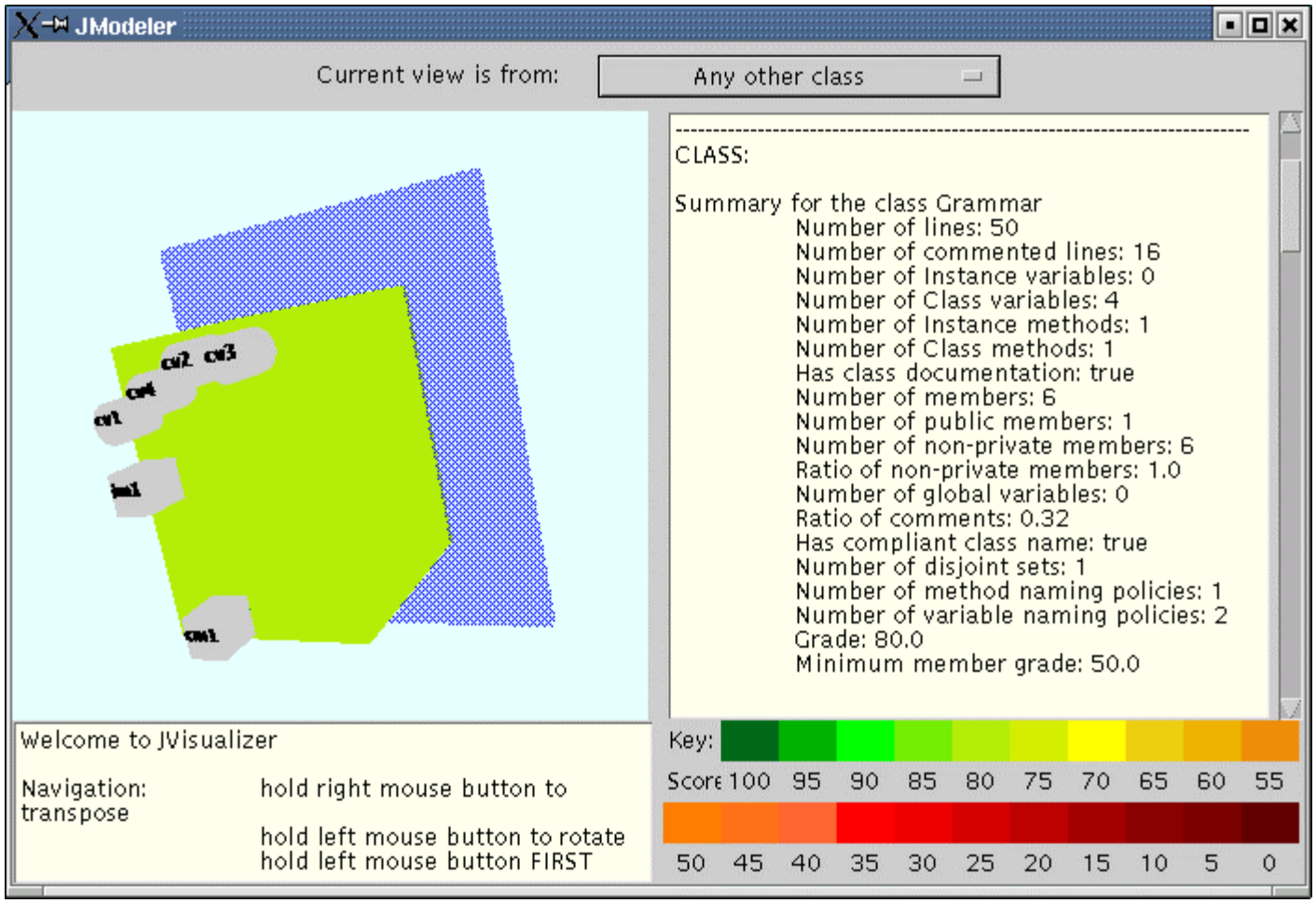

Figure 5-6: The application GUI

CHAPTER 6
RESULTS AND DISCUSSION

Class Grades and Runtime

There were thirteen input programs used in the application prototype. The class grade received by each input program ranged from a low of 5 to a high of 100. This shows that the input programs were composed of a wide range of degree of compliance with the selected programming guidelines. The average class grade was a 71 out of 100, while the mode was an 85. Table 6-1 shows the class grade for each program, as well as the lowest-scoring member within the class, and the time each input program took to parse, analyze, and draw the visualization.

As can be seen in the case of NotaryPublic.java and website.java, the lowest member grade can be higher than the class grade. This is because the class grade is derived from different guidelines than the grades of its variables and methods. Refer to Table 5.2 for which guidelines were used to assess each Java element.

Table 6-1 also shows that no program took more than 2 seconds to run. One of the programs, TinyParser.java, contains 1,430 lines of code, so it is expected that most Java programs used as input to this application can be drawn in less than 2 seconds.

Table 6-1: Overall grading and runtime results

Input Program	Class Grade	Minimum Member Grade	Application Runtime (ms)
BigCube.java	95	50	966
Grammar.java	85	50	966
HazardChecker.java	75	60	1121
Hazard.java	85	65	1055
HtmlLink.java	85	80	1483
IWonEvent.java	70	50	1399
NapsterSong.java	85	80	1508
NotaryPublic.java	55	65	1477
Scanner.java	100	60	1601
ScoreBoardSimulator.java	35	30	1619
StripQualifiers.java	75	70	984
TinyParser.java	75	65	1676
website.java	5	50	1433

Notes: The lowest and highest class grades are shaded. The application was run on a Sony Vaio PCG-GR170k with a Pentium III 1GHz 133 MHz FSB, 256MB RAM, ATI Radeon Mobility Video Card, DRI rendering.

Overview of the Visualization

To save physical space this section shows snapshots of only the visualization portion of the application's GUI. Please refer to Figure 5-6 for a snapshot of the entire GUI. The input program BigCube.java is used in this section to explain the visualization and how the user interacts with it. Refer to Appendix C for screen captures for all the input programs.

When the visualization is first drawn, the class appears as a hexahedron seen head-on, as shown in Figure 6-1. The color of the class' skin, which can be compared to the color legend (see Figure 5-6), reveals its grade (a 95 in BigCube.java's case).

This starting view is from "any other class" as the pull-down menu at the top of the GUI reads. What this means is that all the class' members that are accessible from a class that is not necessarily a subclass of this class nor in the same package as this class, are visible, i.e., not blacked out. Figure 6-1 shows three blacked-out variables (displayed

as cylinders) that are not visible from 'any other class'. However, three methods (displayed as hexahedrons) are visible. The visible members are labeled with a prefix, "cm" for class method, "im" for instance method, "cv" for class variable, and "iv" for instance variable. Following the prefix is a unique number which is assigned in the order that they are encountered in the source file. A blue 'fence', which will be discussed later in this section, can be seen behind the class.

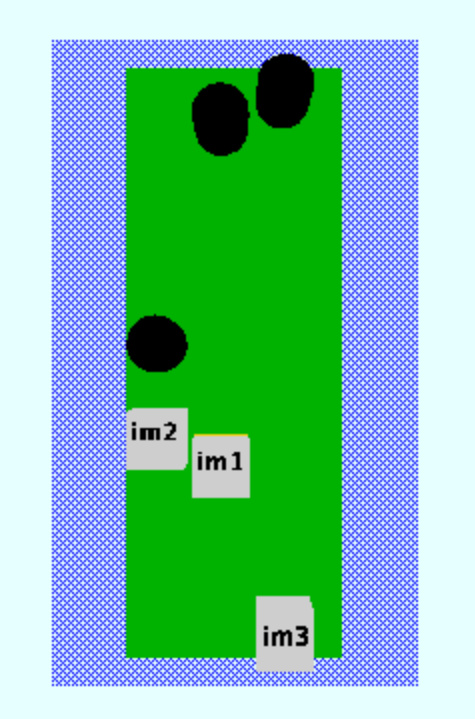

Figure 6-1: The initial graphic displayed for the visualization of BigCube.java
Notes: The three blacked out class or instance variables in this class are not visible from the current viewing perspective (from an unrelated class) because of how they were declared in the class (either "private" or "protected"). Three instance methods are visible, however.

Changing the viewing perspective by using the pull-down menu at the top of the GUI to "Within the class" makes the class "skin" transparent, as shown in Figure 6-2. From within the class all the members are visible no matter how they are declared. Each member's skin color reveals its grade using the scale grading scale as was used on the

class. In the case of Figure 6-2, there are two class variables, "cv1" and "cv2", which have fairly low grades, as can be seen by their orange color. Recall from Chapter Five that a traffic light metaphor was selected for use in the visualization encoding. The closer a member's color is to green, the higher the grade. The closer a member's color is to red, the lower the grade. The exact grading scale used can be seen in Appendix C, Figure C-1.

Also visible in Figure 6-2 are the message calls between the class or instance methods and variables. Each line represents all the calls between two members. The thicker the line, the more messages are sent between the two members. The class or instance variables (displayed as cylinders) that are used the most by the methods are positioned closer to the methods (displayed as hexahedrons). Similarly, the methods that send the most messages within the class, or that are sent the most messages by other methods in the class are positioned closer to the variables. The result is that there is a zone of the most heavily used members within the center of each class visualization. This reveals the members in the most cohesive core of the class, and it can be inferred that these members are the most essential members of this class. Members further from this zone can be considered less critical to the design and function of this class.

All navigation of the visualization is done with the user's mouse. When the user holds down their left mouse button and drags their mouse on the visualization, it rotates in the drag direction. When the user holds down the right mouse button and drags on the visualization, it transposes with the drag. When the user holds down the left mouse button and the "Alt" key and drags their mouse up, the visualization zooms out; if they drag down it zooms out.

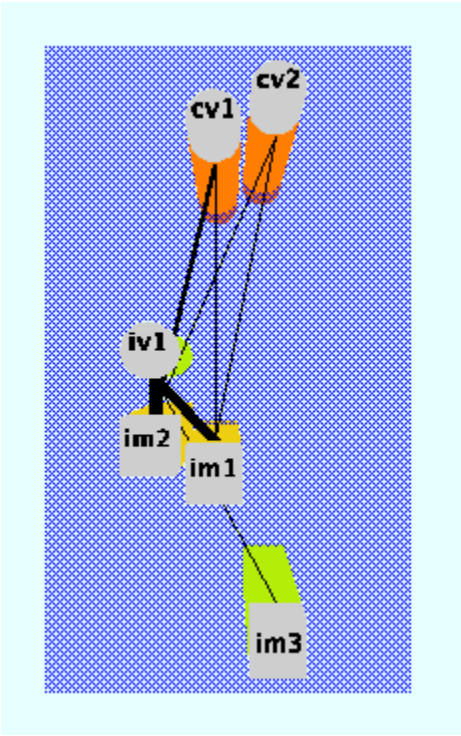

Figure 6-2: A view of BigCube.java from within the class
Notes: All members are visible within the class, therefore none are blacked out. this class has one instance variable ("iv1"), two class variables ("cv1", "cv2"), three instance methods ("im1", "im2", "im3"), and no class methods.

Rotating the class around the y axis reveals a view of the class from behind the blue fence, as shown in Figure 6.3. The blue fence is another technique used in this visualization to help the user quickly gauge the grade of the class or its members. When a class or a class member has a relatively good score, it will not come touch the fence. Only the low-scoring elements (75 and below, out of 100) pass through it.

Discussion of Visualization

Presentation of Quality

During the development of JVisualizer, three presentation techniques were tried to show the quality of a Java element: color, light, and the fence metaphor. Both color and the fence metaphor were found to clearly present the quality of a Java element. Compare the two classes shown in Figure 6-4. The class on the left received a 95, the

class on the right received a 5. The fence metaphor and the colors make this difference clear. Also observe in Figure 6-5 that even when a class has many members the members with low scores 'stick out'. Light, however, was not successful for several reasons.

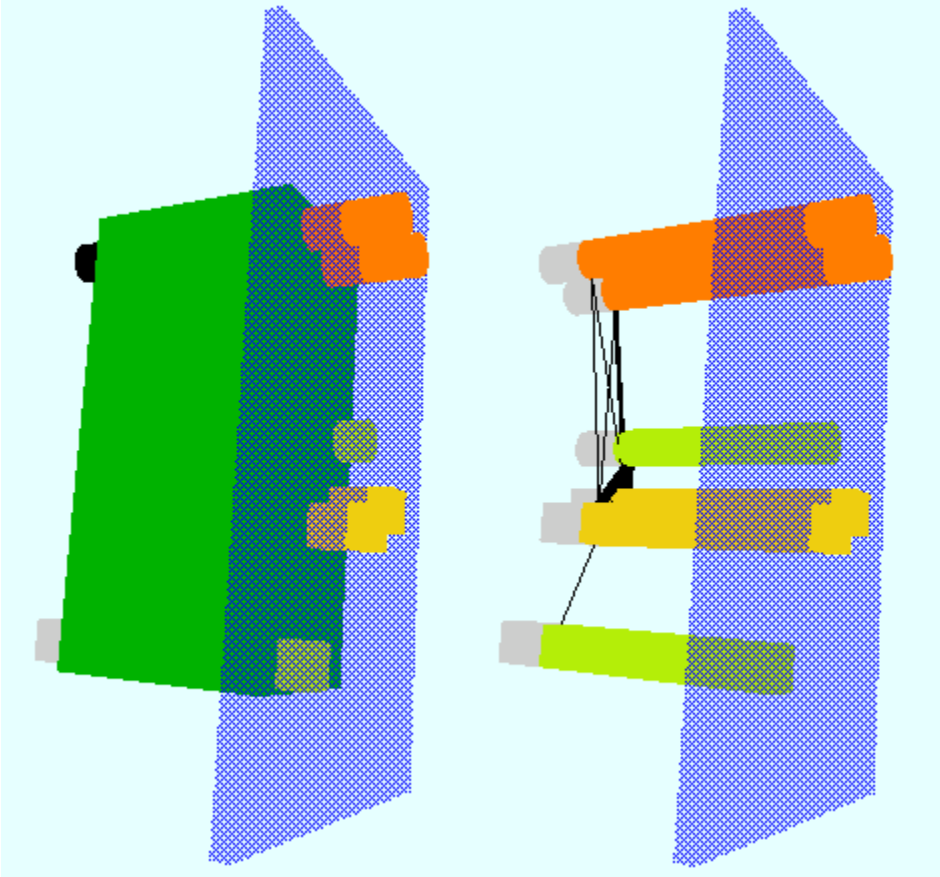

Figure 6-3: Two views of BigCube.java from behind the fence
Notes: On the left is a view from 'any other class'; the right shows a view from 'within the class'. Members that stick out of the fence have low grades. In this case there are two variables (orange cylinders), which appear to be the lowest-scoring members, and two methods (yellow hexahedrons), which were low-scoring, but not as low as the variables.

It was thought that either 'spotlights' could shine on the lowest-scoring elements, or that these same elements could appear to glow. When a spotlight (either a DirectionalLight or SpotLight in the Java 3D API) shines on an element, the effect is that the colors on every face of the element become different. There would be no way to identify the grade of an element by the color anymore. When an element is given a material with emissive qualities to make it seem like it is glowing from within, the only

effect is to change the color of element to the color of the emissive light. There are no 'glow effects' in the space around the element so the effect is lost. Until lighting in 3D graphics becomes more like lighting in 'real life', it is difficult to use it to emphasize parts of a visualization.

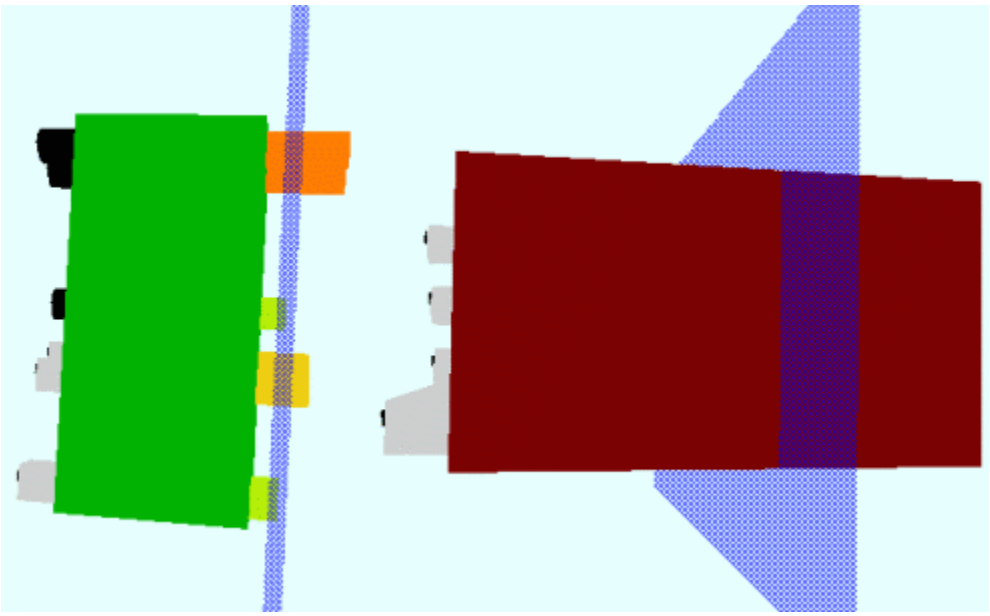

Figure 6-4: Successful presentation techniques for showing quality: color and the fence metaphor
Notes: The visualized class on the left received a 95; the one on the right received a 5.

Presentation of Visibility

Two presentation techniques were tried to show the visibility of Java members: transparency and the 'blacking out' of the labels. While both worked technically, the blacking out technique had additional benefits besides being able to show visibility. The blacked-out members are placeholders in the class visualization. They show that there are members that are hidden. It is as though transparent members do not exist at all. The blacking-out technique reveals patterns among the classes.

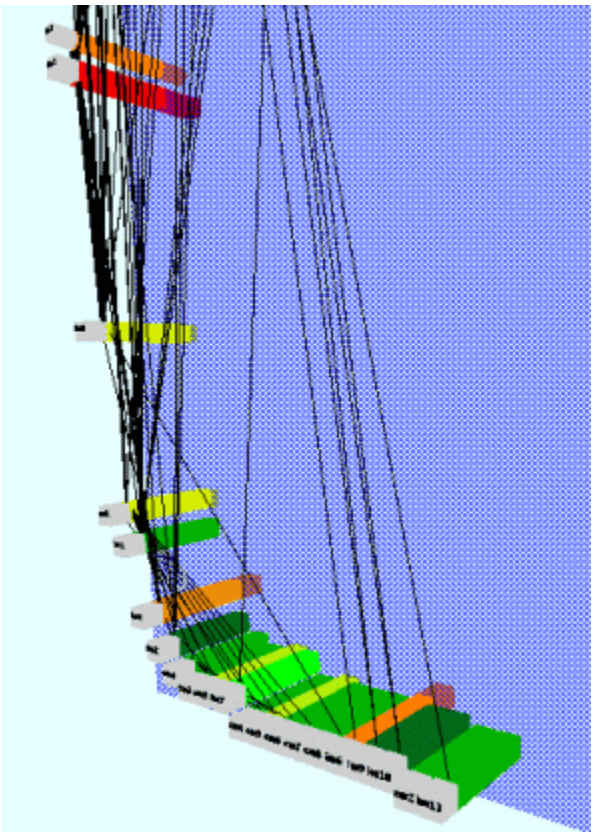

Figure 6-5: The lowest-scoring elements (those with colors closest to red) stand out.

Observe Figure C-5 in Appendix C. It shows that all of Hazard.java's class and/or instance variables are hidden from unrelated classes. This is considered good programming practice. If they had been transparent the information-hiding would not be apparent.

A different pattern shows up in Figures C-4, C-11 and C-13 in Appendix C. These classes contain many private methods. When the source code for these three programs is examined, we can see that these three programs are largely procedural, as opposed to being used as data structures. They contain many private 'helper' methods to conduct some overall function. This is also is considered good programming practice to hide methods that are only used internally.

Presentation of Cohesion

The presentation techniques that were tried to show the cohesion within the class were position and lines with varying width. Both techniques were found to have advantages and disadvantages.

The height of the class visualization is dependent on the range of cohesion within the class. An example will illustrate this. Say that a class has a method that makes a total of 80 calls to class or instance variables within the class, and that this method makes more of calls than any other method in the class. Say that there is another method in the class that makes a total of one call to a class or instance variable (or method) within the class. The call range among these methods is 81, and the result is that JVisualizer will make the class very tall so that it can show the difference in cohesion between the methods. When the range is high as in this example, the class becomes too tall to view it in its entirety. An examples of a class like this is shown in Figures C-10.

One solution to this problem is to pick a maximum class height and then scale down if the class height exceeds the maximum. This alternative was tried but rejected because it was felt that there may be currently-unknown advantages to highlighting the cohesion disparity in these classes. If nothing else it shows how highly-used particular members are as compared to other members and the relative effect on the class of altering or removing these elements.

Using the lines of varying widths makes obvious how tightly-coupled certain members are. The disadvantage is that it is difficult to lay these lines out so that they do not intersect members that lie in-between the two members. The way JVisualizer lays out the members minimizes this problem but does not solve it. By arranging the members

that will have the most lines attached to them to the left and closer to the center of the class means that there are fewer members to get in the way of their lines.

The lines bring out some interesting patterns. Figures C-5 and C-8 show a very regular, ordered sequence of lines extending from the methods to the variables. These are "getter" methods.

Another pattern brought out by the lines is illustrated in Figure 6-6, taken from the visualization of TinyParser.java. Notice that the two instance methods (displayed as hexahedrons toward the left-center of the image) are being called by other methods more than the variables above them are. This pattern turned up in the programs that were more procedural than data structures.

Future Research

JVisualizer is successful at certain things, such as making clear which elements in a Java program need the most improvement based on programming guidelines that are customizable. However, it leaves room for a tremendous amount of work.

JVisualizer only works with single-classes. It does not try to draw multi-class programs. Incorporating this ability into the application would require many presentation decisions, such as how to differentiate between the user's classes and other classes like those provided by Sun. There would also need to be a way to expand and contract the class hierarchy because of the screen 'real estate problem' common in visualizations.

Another challenge would be to try to distinguish between different "families" of classes, such as those in the same package, or classes that are siblings of a superclass. A similar problem is how to differentiate between class members and instance members. This would be the natural step for JVisualizer to take as this is still a class-level problem,

and beginning Java programmers need to understand the difference between elements that are created one time and those that are created every time a class object is instantiated.

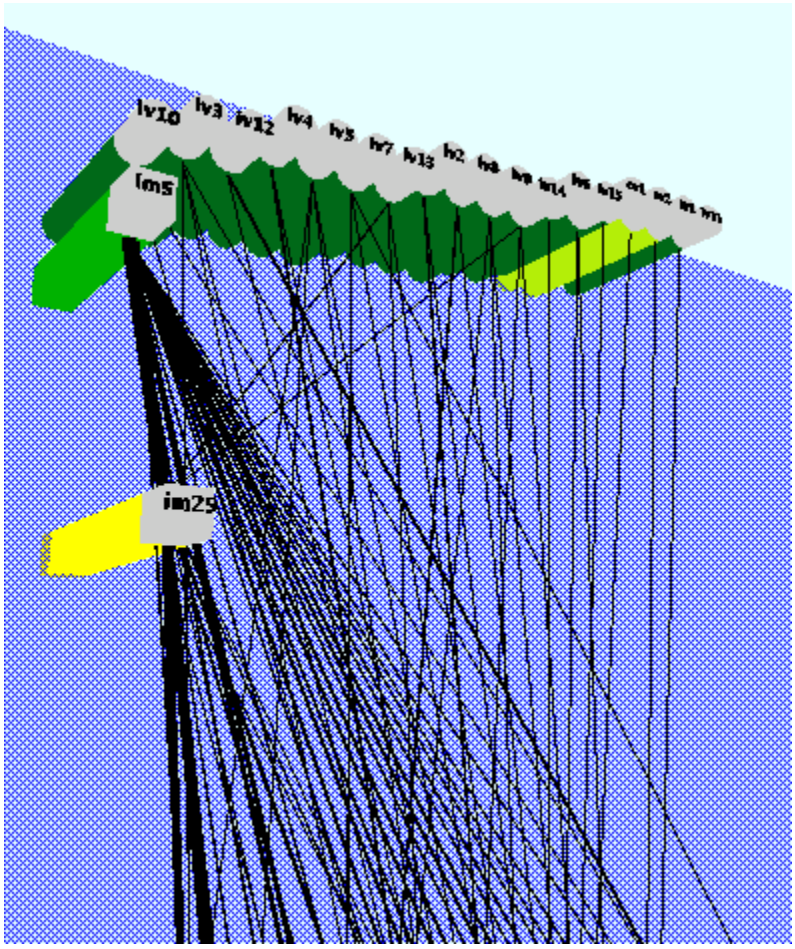

Figure 6-6: The two frequently-called methods and rarely-called variables show that this class is more procedural than it is a data structure

Finally, there is the non-visualization work that JVisualizer's results bring forth. The guidelines literature states that cohesion within a class is desirable, in part because it means that all the elements within the class belong there, i.e. the class was designed well. However, what should this cohesion "look" like? Should it have a small range between the cohesion values or does it depend on the type of class? It may be that JVisualizer's most important contribution is that it shows that different types of classes look different

from each other. This implies that these different types of classes need to be identified. It might be that there should be a common core of programming guidelines for Java, but that there should be different 'extensions' of guidelines based on the purpose of the class.

APPENDIX A
JVISUALIZER SOURCE CODE

JCallMatrix.java

```
/**
 * JCallMatrix
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JCallMatrix {

    /* Instance Variables */
    private JGrammar grammar; // can recognize Java grammar (types, modifiers, etc.)
    private int matrixHeight;
    private int matrixWidth;
    private int [] [] uses; // a count of the calls of each internal class/instance
                            // method or variable
    private Vector used; // all the names of the class/instance vars and methods

    /* Public Methods */
    public JCallMatrix() {
        this.grammar = new JGrammar();
        this.uses = null;
        this.used = null;
    }

    /**
     * Creates a matrix for a JClass object of call/use of variables and methods by
     *  methods.
     * Organized as such:
     *
     *                                      CALLEES:
     *                      class_meths  inst_meths  class_vars  inst_vars
     * CALLERS:            ---------------------------------------------
     *       class_meths |             |          |           |          |
     *                     ---------------------------------------------
     *       inst_meths  |             |          |           |          |
     *                     ---------------------------------------------
     *
     * Each matrix cell stores the number of times that the caller calls the callee.
     *
     * @param          theClass       Java Class to visualize
     */
    public JCallMatrix buildMatrix(JClass theClass) {
        System.out.println("\tBuilding the call/use matrix...");

        int width = -1; // matrix columns
        int height = -1; // matrix rows

        height = theClass.getNClassMethods() + theClass.getNInstanceMethods();
        this.matrixHeight = height;
        width = theClass.getNClassVariables() + theClass.getNInstanceVariables() +
                height;
        this.matrixWidth = width;
        this.uses = new int [height] [width];

        /* add the names of the class/instance vars and methods to used vector */
        this.buildUsedVector(width, theClass);

        /* initialize each matrix cell to 0 */
        for (int i=0; i<height; i++) {
            for (int j=0; j<width; j++) {
                this.uses[i][j] = 0;
            }
```

```
    }

    /* count the method and variable use by class methods */
    JClassMethod jcm;
    for (int i=0; i<theClass.getNClassMethods(); i++) {
        jcm = theClass.getClassMethod(i);
        this.countUses(jcm, theClass);
    }

    /* count the method and variable use by instance methods */
    JInstanceMethod jim;
    for (int i=0; i<theClass.getNInstanceMethods(); i++) {
        jim = theClass.getInstanceMethod(i);
        this.countUses(jim, theClass);
    }

    /* sum the number called from and to for each class/instance
     * method and variable */
    this.sumUses(width, height, theClass);

    return this;
}

/**
 * Fills the used vector with the names of all the classes elements in
 * this order:
 * class methods, instance methods, class variables, instance variables
 *
 * @param         vectorLength   the number of methods and variables
 * @param           theClass        a JClass object
 */
private void buildUsedVector(int vectorLength, JClass theClass) {
    this.used = new Vector(vectorLength);

    /* add the class method names */
    JClassMethod jcm;
    for (int i=0; i<theClass.getNClassMethods(); i++) {
        jcm = theClass.getClassMethod(i);
        this.used.addElement(jcm.getMethodName());
    }

    /* add the instance method names */
    JInstanceMethod jim;
    for (int i=0; i<theClass.getNInstanceMethods(); i++) {
        jim = theClass.getInstanceMethod(i);
        this.used.addElement(jim.getMethodName());
    }

    /* add the class variable names */
    JClassVariable jcv;
    for (int i=0; i<theClass.getNClassVariables(); i++) {
        jcv = theClass.getClassVariable(i);
        this.used.addElement(jcv.getVariableName());
    }

    /* add the instance variable names */
    JInstanceVariable jiv;
    for (int i=0; i<theClass.getNInstanceVariables(); i++) {
        jiv = theClass.getInstanceVariable(i);
        this.used.addElement(jiv.getVariableName());
    }
}

/**
 * For a single method it counts the calls/uses of all other class/instance
 * methods and vars in that method, and adds this to the call/use matrix
 *
 * @param         theMethod      JClassMethod or JInstanceMethod object
 * @param           theClass        a JClass object that contains the JMethod
 */
private void countUses(JMethod theMethod, JClass theClass) {
    Vector localVars = new Vector(20);
    boolean isOuterInstanceMethod = false; // differentiates an instance method
                                           // with same name as one in the class
                                           // and one applying to a local var
    boolean isThis = false; // differentiates scopes (Ex: this.x = x)
                            // true for the first x but not the second
    boolean pastMethodDeclaration = false;
    boolean pastParameters = false;
    boolean wasNew = false; // if last non-blank word was "new"
                            // used so that a object decln does not count the
```

```
                        // use twice (Ex: TheClass c = new TheClass)
boolean wasType = false; // if last non-blank token was a type
int matrixRow = this.used.indexOf(theMethod.getMethodName());
int matrixCol = -1;
//System.out.println("\n" + theMethod.getMethodName());

// loop through each line of the method
for (int i=0; i<theMethod.getNSourceLines(); i++) {
    String trimLine = theMethod.getSourceLine(i).trim();
    StringTokenizer st =
        new StringTokenizer(trimLine, " \t(){}[],;=><.+-:!/\"", true);
    String token = null;
    String lastToken = "-1";
    String lastLastToken = "-2";
    boolean inString = false;  // notes when in the middle of a string
    boolean isEscapedQuote = false; // Ex: \"
    int nQuote = 0; // number of double quotes seen on the line

    // loop through each token looking for a match into the used vector
    while (st.hasMoreTokens()) {
        token = st.nextToken();
        isEscapedQuote = false;

        if (lastLastToken.equals("/") && (lastToken.equals("/") ||
                                          lastToken.equals("*"))) {
            break;
        }

        // will not find any class/instance variable or method use
        // until its past the method declaration
        if (token.equals("(")) {
            pastMethodDeclaration = true;
        } else if (token.equals(")")) {
            pastParameters = true;
        }

        if (lastLastToken.equals("\\") && lastToken.equals("\"")) {
            isEscapedQuote = true;
            //System.out.println("Is escaped quote.");
            inString = true;
        }

        if (token.equals("\"") && !isEscapedQuote) {
            nQuote++;
        }

        if (nQuote % 2 == 1) {
            inString = true;
            //System.out.println("In string on token: " + token +
            //" nQuote: " + nQuote);
        } else {
            inString = false;
            //System.out.println("Not in string on token: " + token +
            //" nQuote: " + nQuote);
        }

        // need to know if in a this.* situation to know the scope
        if (lastLastToken.equals("this") && lastToken.equals(".")) {
            isThis = true;
            //System.out.println("In a this");
        } else {
            isThis = false;
        }

        // need to know if this is a localVar.<instMethod> situation
        // to know the scope
        if (localVars.contains(lastLastToken) && lastToken.equals(".")) {
            isOuterInstanceMethod = true;
            //System.out.println("In a outer inst meth situation.");
        } else {
            isOuterInstanceMethod = false;
        }

        // remember the use of the 'new' keyword so that a class
        // use is not overcounted
        if (lastToken.equals("new")) {
            wasNew = true;
        } else {
            wasNew = false;
        }
```

```
// remember having  seen types - this is the beginning of a
// local variable declaration
if (this.grammar.isType(lastToken) ||
    this.grammar.isJDKClass(lastToken)) {
    wasType = true;
} else {
    wasType = false;
}

// check if a local variable can be remembered
if (wasType && !token.equals(" ") && pastMethodDeclaration &&
               !token.equals("(") && !token.equals("[")) {
    if (!localVars.contains(token)) {
        // found a new local variable
        localVars.addElement(token);
        //System.out.println("Added " + token + " to localVars.");
    }
}

// toss out tokens that are definitely not identifiers
if (this.grammar.isType(token) || this.grammar.isModifier(token) ||
    this.grammar.isKeyword(token) ||
    this.grammar.isOperator(token) ||
    this.grammar.isJDKClass(token) ||
    this.grammar.isNumber(token) ||
    !pastMethodDeclaration || token.equals("[") ||
    token.equals("]") || token.equals("(") || token.equals(" ") ||
    token.equals(")") || token.equals("{") || token.equals("}") ||
    token.equals("\t") || token.equals(";") || token.equals(".")) {
        // ignore token
} else if (!localVars.contains(token) || isThis) {
    if (token.equals(theClass.getClassName()) && !wasNew &&
                                                  !inString) {
        // Found a constructor use - now need to start swallowing
        // parameters so that we can figure out which constructor
        // is being used
        String newName = token.concat("_"); // could leave this
                                            // for input progs
        for (int j=0; j<this.used.size(); j++) {
            if (newName.equals(this.used.elementAt(j))) {
                matrixCol = this.used.indexOf(newName);
                this.uses[matrixRow][matrixCol]++;
                //System.out.println("Found a match with " +
                //    newName);
                //System.out.println("Incremented " + matrixRow +
                //    " " + matrixCol);
            }
        }
    } else if (!wasNew && !token.equals(theMethod.getMethodName())
                       && pastParameters &&
                       !isOuterInstanceMethod && !inString) {
        // have to search the store
        for (int j=0; j<this.used.size(); j++) {
            if (token.equals(this.used.elementAt(j))) {
                matrixCol = this.used.indexOf(token);
                this.uses[matrixRow][matrixCol]++;
                //System.out.println("Found a match with " + token);
                //System.out.println("Incremented " + matrixRow +
                //    " " + matrixCol);
            }
        }
    } else if (!wasNew && !token.equals(theMethod.getMethodName())
                       && !pastParameters) {
        // may be a local var (also a param) of non-JDK class
        for (int j=0; j<this.used.size(); j++) {
            if (token.equals(this.used.elementAt(j))) {
                if (!localVars.contains(token)) {
                    // found a new local variable
                    localVars.addElement(token);
                    //System.out.println("Added " + token +
                    //    " to localVars.");
                }
            }
        }
    }
}

// remember last non-blank token to use to identify local
// variables
if (!token.equals(" ")) {
    lastLastToken = lastToken;
```

```
                lastToken = token;
            }
        }
    }
}

public Vector getAllUsed() {
    return this.used;
}

public int[][] getAllUses() {
    return this.uses;
}

public int getMatrixHeight() {
    return this.matrixHeight;
}

public int getMatrixWidth() {
    return this.matrixWidth;
}

public String getUsed(int index) {
    return this.used.elementAt(index).toString();
}

public int getUses(int row, int column) {
    return this.uses[row][column];
}

/**
 * Sums the uses from and to class/instance methods and variables and assigns
 * these statistics to the methods and variables
 *
 * @param           theClass        a JClass object that contains the JMethod
 */
private void sumUses(int matrixWidth, int matrixHeight, JClass theClass) {
    int nCallsFrom = 0;
    int nCallsTo = 0;
    JClassMethod jcm = null;
    JInstanceMethod jim = null;
    String memberType = null; // "method", "variable", or null
    String methodName = null;
    String methodType = null; // "class", "instance", or null
    String variableName = null;
    String variableType = null; // "class", "instance", or null

    /* sum up the matrix rows which are the calls from methods */
    for (int i=0; i<matrixHeight; i++) {
        nCallsFrom = 0;

        // traverse every class/instance method or variable
        for (int j=0; j<matrixWidth; j++) {
            nCallsFrom += this.uses[i][j];
        }

        // assign the calls-from to the applicable method
        methodName = (String) this.used.elementAt(i);
        methodType = theClass.getMethodType(methodName);
        theClass.setCallsFrom(methodName, methodType, nCallsFrom);

        //System.out.println("Total uses by method in row " + i + ": " +
        //    nCallsFrom);
    }

    /* reset reused variable names */
    methodName = null;
    methodType = null;

    /* sum up the matrix columns which are the calls to methods/variables */
    for (int j=0; j<matrixWidth; j++) {
        nCallsTo = 0;

        // traverse every method
        for (int i=0; i<matrixHeight; i++) {
            nCallsTo += this.uses[i][j];

            if (j < matrixHeight) {
                // summing for a method
                // (The height of the matrix is equal to the the width of the
                // methods portion of the width)
```

```
                    memberType = "method";
                } else {
                    // summing for a variable
                    memberType = "variable";
                }
            }

            // assign the calls-to to the applicable method/variable
            if (memberType.equals("method")) { // class or instance method
                methodName = (String) this.used.elementAt(j);
                methodType = theClass.getMethodType(methodName);
                theClass.setCallsTo(methodName, methodType, nCallsTo, memberType);
            } else { // class or instance variable
                variableName = (String) this.used.elementAt(j);
                variableType = theClass.getVariableType(variableName);
                theClass.setCallsTo(variableName, variableType, nCallsTo,
                                    memberType);
            }
        }
    }
}
```

JClass.java

```
/**
 * JClass
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JClass {

    /* Instance Variables */
    private boolean classDoc; // existence of a class documentation header
    private Vector classMethods; // all class methods (JClassMethod)
    private String className; // the class name
    private String classPackage; // the class package name if applicable
    private Vector classVariables; // all class variables (JClassVariable)
    private Vector criticism; // broken guidelines of class as Strings
    private double grade;
    private boolean hasCompliantName; // if name is compliant
    private Vector instanceMethods; // all instance methods (JInstanceMethod)
    private Vector instanceVariables; // all instance variables (JInstanceVariable)
    private double minimumGrade;
    private Vector minimumGradeMembers;
    private int nCommentLines; // number of commented lines
    private int nDisjointSets; // number of non-cohesive sets of class/instance
                               // methods and vars
    private int nGlobalVariables; // number of public non-final class variables
    private int nMethNamingPolicies; // number of method naming policies used
    private int nNotPrivateMembers; // number of non-private class/instance
                                    // methods and variables
    private int nPublicMembers; // number of public class/instance methods and vars
    private int nMembers; // number of class/instance methods and variables
    private int nSourceLines; // the number of source lines in outer-most scope
    private int nVarNamingPolicies; // number of variable naming policies used
    private double rComments; // comment density
    private double rNotPrivateMembers; // ratio (0.0-1.0) of non-private
                                       // class/instance methods and variables
    private Vector sortedMethodCalls;
    private Vector sortedVariableCalls;

    /* Public Methods */

    /**
     * Class constructor
     */
    public JClass() {
        this.classDoc = false;
        this.classMethods = new Vector(30);
        this.className = null;
        this.classPackage = null;
        this.classVariables = new Vector(30);
        this.criticism = new Vector(10);
```

```
    this.grade = 100.0;
    this.hasCompliantName = false;
    this.instanceMethods = new Vector(30);
    this.instanceVariables = new Vector(30);
    this.minimumGrade = 100.0;
    this.minimumGradeMembers = null;
    this.nCommentLines = 0;
    this.nDisjointSets = 0;
    this.nGlobalVariables = 0;
    this.nMembers = 0;
    this.nMethNamingPolicies = 0;
    this.nNotPrivateMembers = 0;
    this.nPublicMembers = 0;
    this.nSourceLines = 0;
    this.nVarNamingPolicies = 0;
    this.rComments = 0.0;
    this.rNotPrivateMembers = 0.0;
    this.sortedMethodCalls = null;
    this.sortedVariableCalls = null;
}

/**
 * Adds a class method to the class
 *
 * @param         name           method name
 * @param         visibility     "public","private","protected","default"
 * @param           isFinal        if it is declared "final"
 * @param           foundDoc       if it had method documentation
 */
public void addClassMethod(String name, String visibility, boolean isFinal,
                           boolean foundDoc){
    String label = "cm" + (this.getNClassMethods() +1);
    this.classMethods.addElement(new JClassMethod(name,visibility,isFinal,
                                                  foundDoc,label));
    this.nMembers++;
    if (visibility.equals("public")) {
        this.nPublicMembers++;
    }
    if (!visibility.equals("private")) {
        this.nNotPrivateMembers++;
    }
}

/**
 * Adds a class variable to the class
 *
 * @param           name          method name
 * @param         visibility     "public","private","protected","default"
 * @param           isFinal        if it is declared "final"
 * @param           foundDoc       if it had method documentation
 */
public void addClassVariable(String name, String visibility, boolean isFinal,
                             boolean foundDoc){
    String label = "cv" + (this.getNClassVariables() +1);
    this.classVariables.addElement(new JClassVariable(name,visibility,isFinal,
                                                      foundDoc, label));
    this.nMembers++;
    if (visibility.equals("public")) {
        this.nPublicMembers++;
    }
    if (!visibility.equals("private")) {
        this.nNotPrivateMembers++;
    }
}

public void addCriticism(String criticism) {
    this.criticism.addElement(criticism);
}

/**
 * Adds an instance method to the class
 *
 * @param         name           method name
 * @param         visibility     "public","private","protected","default"
 * @param           foundDoc       if it had method documentation
 * @param           isConstructor   if it is a constructor method
 */
public void addInstanceMethod(String name, String visibility, boolean foundDoc,
                              boolean isConstructor){
    String label = "im" + (this.getNInstanceMethods() +1);
    this.instanceMethods.addElement(new JInstanceMethod(name, visibility,
```

```
                                    foundDoc, isConstructor, label));
    this.nMembers++;
    if (visibility.equals("public")) {
        this.nPublicMembers++;
    }
    if (!visibility.equals("private")) {
        this.nNotPrivateMembers++;
    }
}

/**
 * Adds an instance variable to the class
 *
 * @param         name           method name
 * @param         visibility     "public","private","protected","default"
 * @param           foundDoc       if it had method documentation
 */
public void addInstanceVariable(String name, String visibility,
                                boolean foundDoc){
    String label = "iv" + (this.getNInstanceVariables() +1);
    this.instanceVariables.addElement(new JInstanceVariable(name,visibility,
                                      foundDoc,label));
    this.nMembers++;
    if (visibility.equals("public")) {
        this.nPublicMembers++;
    }
    if (!visibility.equals("private")) {
        this.nNotPrivateMembers++;
    }
}

/**
 * Add a line of source code to a method
 *
 * @param         line           line of source code
 * @param         methodType     "class", "instance"
 */
public void addMSourceLine(String line, String methodType) {
    int index = -1;

    if (methodType.equals("class")) {
        index = this.classMethods.size()-1;
        JClassMethod jcm = (JClassMethod) this.classMethods.elementAt(index);
        jcm.addSourceLine(line);
        this.classMethods.set(index, jcm);
    } else if (methodType.equals("instance")) {
        index = this.instanceMethods.size()-1;
        JInstanceMethod jim = (JInstanceMethod)
                              this.instanceMethods.elementAt(index);
        jim.addSourceLine(line);
        this.instanceMethods.set(index, jim);
    } else {
        System.out.println("Error: methodType: " + methodType);
    }
}

/**
 * Used to add params to the end of a constructor name when it had a multi-line
 * method declaration
 *
 * @param         addition       additional parameters to add to the name
 */
public void addToMethodName(String addition) {
    int index = -1;
    String newName;

    index = this.instanceMethods.size()-1;
    JInstanceMethod jim = (JInstanceMethod)
                          this.instanceMethods.elementAt(index);
    newName = jim.getMethodName().concat(addition);
    jim.setMethodName(newName);
    this.instanceMethods.set(index, jim);
}

public JClassMethod getClassMethod(int index) {
    return (JClassMethod) this.classMethods.elementAt(index);
}

public String getClassName() {
    return this.className;
}
```

```
public JClassVariable getClassVariable(int index) {
    return (JClassVariable) this.classVariables.elementAt(index);
}

public Vector getCriticism() {
    return this.criticism;
}

public double getGrade() {
    return this.grade;
}

public boolean getHasCompliantName() {
    return this.hasCompliantName;
}

/**
 * Returns the index of the instance or class method in the class' internally
 * stored data structure
 *
 * @param         methodType     "class" or "instance"
 * @param          methodName      name of method
 * @return         int             the index
 */
public int getIndexOfMethod(String methodType, String methodName) {
    int index = -1;

    if (methodType.equals("class")) {
        JClassMethod jcm = null;
        for (int i=0; i<this.getNClassMethods(); i++) {
            jcm = this.getClassMethod(i);
            if (jcm.getMethodName().equals(methodName)) {
                index = i;
            }
        }
    } else if (methodType.equals("instance")) {
        JInstanceMethod jim = null;
        for (int i=0; i<this.getNInstanceMethods(); i++) {
            jim = this.getInstanceMethod(i);
            if (jim.getMethodName().equals(methodName)) {
                index = i;
            }
        }
    } else {
        System.out.println("Error - not a valid method type: " + methodType);
    }

    return index;
}

/**
 * Returns the index of the instance or class variable in the class' internally
 * stored data structure
 *
 * @param         variableType     "class" or "instance"
 * @param          variableName      name of variable
 * @return         int               the index
 */
public int getIndexOfVariable(String variableType, String variableName) {
    int index = -1;

    if (variableType.equals("class")) {
        JClassVariable jcv = null;
        for (int i=0; i<this.getNClassVariables(); i++) {
            jcv = this.getClassVariable(i);
            if (jcv.getVariableName().equals(variableName)) {
                index = i;
            }
        }
    } else if (variableType.equals("instance")) {
        JInstanceVariable jiv = null;
        for (int i=0; i<this.getNInstanceVariables(); i++) {
            jiv = this.getInstanceVariable(i);
            if (jiv.getVariableName().equals(variableName)) {
                index = i;
            }
        }
    } else {
        System.out.println("Error - Not a valid variable type: " +
                           variableType);
```

```
    }

    return index;
}

public JInstanceMethod getInstanceMethod(int index) {
    return (JInstanceMethod) this.instanceMethods.elementAt(index);
}

public JInstanceVariable getInstanceVariable(int index) {
    return (JInstanceVariable) this.instanceVariables.elementAt(index);
}

/* returns the member label
 * looks like "CM#", "IM#", "CV#", or "IV#"
 * (used for labeling the member in the visualization)
 */
public String getLabel(String memberName) {
    String label = null;
    String type = this.getMemberType(memberName);
    int index = -1;

    if (type.equals("class method")) {
        type = "class";
        index = getIndexOfMethod(type, memberName);
        JClassMethod jcm = this.getClassMethod(index);
        label = jcm.getLabel();
    } else if (type.equals("instance method")) {
        type = "instance";
        index = getIndexOfMethod(type, memberName);
        JInstanceMethod jim = this.getInstanceMethod(index);
        label = jim.getLabel();
    } else if (type.equals("class variable")) {
        type = "class";
        index = getIndexOfVariable(type, memberName);
        JClassVariable jcv = this.getClassVariable(index);
        label = jcv.getLabel();
    } else if (type.equals("instance variable")) {
        type = "instance";
        index = getIndexOfVariable(type, memberName);
        JInstanceVariable jiv = this.getInstanceVariable(index);
        label = jiv.getLabel();
    } else {
        System.out.println("Error in member type: " + type);
    }

    return label;
}

/* returns "class method", "instance method", "class variable",
 * or "instance variable"
 */
public String getMemberType(String memberName) {
    String type;
    String memberType;
    type = this.getMethodType(memberName);
    if (type != null) {
        if (type.equals("class")) {
            memberType = "class method";
        } else {
            memberType = "instance method";
        }
    } else {
        type = this.getVariableType(memberName);
        if (type.equals("class")) {
            memberType = "class variable";
        } else {
            memberType = "instance variable";
        }
    }
    return memberType;
}

/* returns the range between the maximum and minimum number of method
 * calls-from and calls-to
 */
public int getMethodCohesionRange() {
    int range;
    int calls;
    int minCalls = 10000000; // arbitrarily large
    int maxCalls = 0;
```

```
    JClassMethod jcm = null;
    JInstanceMethod jim = null;

    for (int i=0; i<this.getNClassMethods(); i++) {
        calls = 0;
        jcm = this.getClassMethod(i);
        calls += jcm.getNCaller();
        calls += jcm.getNCallee();
        if (calls > maxCalls) {
            maxCalls = calls;
        }
        if (calls < minCalls) {
            minCalls = calls;
        }
    }

    for (int i=0; i<this.getNInstanceMethods(); i++) {
        calls = 0;
        jim = this.getInstanceMethod(i);
        calls += jim.getNCaller();
        calls += jim.getNCallee();
        if (calls > maxCalls) {
            maxCalls = calls;
        }
        if (calls < minCalls) {
            minCalls = calls;
        }
    }

    range = maxCalls - minCalls;
    if (range == -10000000) {range = 0;}
    //System.out.println("Max meth calls: " + maxCalls + " Min meth calls: " +
    //    minCalls + " meth range: " + range);

    if (this.sortedMethodCalls == null) {
        this.sortMethodCalls(minCalls, maxCalls);
    }

    return range;
}

/* return "instance" or "class" */
public String getMethodType(String methodName) {
    String methodType = null;
    JMethod jm = null;
    boolean foundType = false;

    // search the class methods
    for (int i=0; i<getNClassMethods(); i++) {
        if (!foundType) {
            jm = (JMethod) this.classMethods.elementAt(i);
            if (jm.getMethodName().equals(methodName)) {
                methodType = "class";
                foundType = true;
            }
        }
    }

    // search the instance methods
    if (!foundType) {
        for (int i=0; i<getNInstanceMethods(); i++) {
            jm = (JMethod) this.instanceMethods.elementAt(i);
            if (jm.getMethodName().equals(methodName)) {
                methodType = "instance";
                foundType = true;
            }
        }
    }

    return methodType;
}

public double getMinimumGrade() {
    return this.minimumGrade;
}

public Vector getMinimumGradeMembers() {
    return this.minimumGradeMembers;
}

public int getNClassMethods() {
```

```
    return this.classMethods.size();
}

public int getNClassVariables() {
    return this.classVariables.size();
}

public int getNCommentLines() {
    return this.nCommentLines;
}

public int getNDisjointSets() {
    return this.nDisjointSets;
}

public int getNGlobalVariables() {
    return this.nGlobalVariables;
}

public int getNInstanceMethods() {
    return this.instanceMethods.size();
}

public int getNInstanceVariables() {
    return this.instanceVariables.size();
}

public int getNMembers() {
    return this.nMembers;
}

public int getNMethNamingPolicies() {
    return this.nMethNamingPolicies;
}

public int getNNotPrivateMembers() {
    return this.nNotPrivateMembers;
}

public int getNPublicMembers() {
    return this.nPublicMembers;
}

public int getNSourceLines(){
    return this.nSourceLines;
}

public int getNVarNamingPolicies() {
    return this.nVarNamingPolicies;
}

public String getPackageName() {
    return this.classPackage;
}

public double getRComments() {
    return this.rComments;
}

public double getRNotPrivateMembers() {
    return this.rNotPrivateMembers;
}

public Vector getSortedMethodCalls() {
    return this.sortedMethodCalls;
}

public Vector getSortedVariableCalls() {
    return this.sortedVariableCalls;
}

/* returns the range between the maximum and minimum number of variable
 * calls-to
 */
public int getVariableCohesionRange() {
    int range;
    int calls;
    int minCalls = 10000000; // arbitrarily large
    int maxCalls = 0;
    JClassVariable jcv = null;
    JInstanceVariable jiv = null;
```

```
    for (int i=0; i<this.getNClassVariables(); i++) {
        calls = 0;
        jcv = this.getClassVariable(i);
        calls += jcv.getNCallee();
        if (calls > maxCalls) {
            maxCalls = calls;
        }
        if (calls < minCalls) {
            minCalls = calls;
        }
    }

    for (int i=0; i<this.getNInstanceVariables(); i++) {
        calls = 0;
        jiv = this.getInstanceVariable(i);
        calls += jiv.getNCallee();
        if (calls > maxCalls) {
            maxCalls = calls;
        }
        if (calls < minCalls) {
            minCalls = calls;
        }
    }

    range = maxCalls - minCalls;
    if (range == -10000000) {range = 0;}
    //System.out.println("Max var calls: " + maxCalls + " Min var calls: " +
    //    minCalls + " var range: " + range);

    if (this.sortedVariableCalls == null) {
        this.sortVariableCalls(minCalls, maxCalls);
    }

    return range;
}

/* returns "class" or "instance" */
public String getVariableType(String variableName) {
    String variableType = null;
    JVariable jv = null;
    boolean foundType = false;

    // search the class variables
    for (int i=0; i<getNClassVariables(); i++) {
        if (!foundType) {
            jv = (JVariable) this.classVariables.elementAt(i);
            if (jv.getVariableName().equals(variableName)) {
                variableType = "class";
                foundType = true;
            }
        }
    }

    // search the instance variables
    if (!foundType) {
        for (int i=0; i<getNInstanceVariables(); i++) {
            jv = (JVariable) this.instanceVariables.elementAt(i);
            if (jv.getVariableName().equals(variableName)) {
                variableType = "instance";
                foundType = true;
            }
        }
    }

    return variableType;
}

public boolean hasClassDoc(){
    return this.classDoc;
}

public void incrementNCommentLines() {
    this.nCommentLines++;
}

/* add to the number of comment lines found for a method */
public void incrementNMCommentLines(String methodType) {
    int index = -1;

    if (methodType.equals("class")) {
```

```
        index = this.classMethods.size()-1;
        JClassMethod jcm = (JClassMethod) this.classMethods.elementAt(index);
        jcm.incrementNCommentLines();
        this.classMethods.set(index, jcm);
    } else if (methodType.equals("instance")) {
        index = this.instanceMethods.size()-1;
        JInstanceMethod jim = (JInstanceMethod)
                        this.instanceMethods.elementAt(index);
        jim.incrementNCommentLines();
        this.instanceMethods.set(index, jim);
    } else {
        // Error: should not get here
        System.out.println("The method type: " + methodType);
    }
}

/* add to the number of source code lines found for a method */
public void incrementNMSourceLines(String methodType) {
    int index = -1;

    if (methodType.equals("class")) {
        index = this.classMethods.size()-1;
        JClassMethod jcm = (JClassMethod) this.classMethods.elementAt(index);
        jcm.incrementNSourceLines();
        this.classMethods.set(index, jcm);
    } else if (methodType.equals("instance")) {
        index = this.instanceMethods.size()-1;
        JInstanceMethod jim = (JInstanceMethod)
                        this.instanceMethods.elementAt(index);
        jim.incrementNSourceLines();
        this.instanceMethods.set(index, jim);
    } else {
        // Error: should not get here
        System.out.println("Error: The method type: " + methodType);
    }
}

/* add to the number of source code lines found for the class */
public void incrementNSourceLines() {
    this.nSourceLines++;
}

/* assign the total number of internal method and variable calls from a
 * method to that method */
public void setCallsFrom(String methodName, String methodType,
                         int nCallsFrom) {
    int index = -1;
    index = this.getIndexOfMethod(methodType, methodName);

    // assign calls-from to the method
    if (methodType.equals("class")) {
        JClassMethod jcm = null;
        jcm = this.getClassMethod(index);
        jcm.setNCaller(nCallsFrom);
        this.classMethods.set(index, jcm);
    } else if (methodType.equals("instance")) {
        JInstanceMethod jim = null;
        jim = this.getInstanceMethod(index);
        jim.setNCaller(nCallsFrom);
        this.instanceMethods.set(index, jim);
    }
}

/* assign the total number of calls to a method or variable to that method
 * or variable */
public void setCallsTo(String name, String type, int nCallsTo,
                       String memberType) {
    int index = -1;

    // get the index into the method/variable Vector for the method/variable
    if (memberType.equals("method")) {
        index = this.getIndexOfMethod(type, name);
    } else if (memberType.equals("variable")) {
        index = this.getIndexOfVariable(type, name);
    } else {
        System.out.println("Not valid member type: " + memberType);
    }

    // assign calls-to to the variable/method
    if (memberType.equals("method")) { // method
        if (type.equals("class")) {
```

```
            JClassMethod jcm = null;
            jcm = this.getClassMethod(index);
            jcm.setNCallee(nCallsTo);
            this.classMethods.set(index, jcm);
        } else if (type.equals("instance")) {
            JInstanceMethod jim = null;
            jim = this.getInstanceMethod(index);
            jim.setNCallee(nCallsTo);
            this.instanceMethods.set(index, jim);
        }
    } else { // variable
        if (type.equals("class")) {
            JClassVariable jcv = null;
            jcv = this.getClassVariable(index);
            jcv.setNCallee(nCallsTo);
            this.classVariables.set(index, jcv);
        } else if (type.equals("instance")) {
            JInstanceVariable jiv = null;
            jiv = this.getInstanceVariable(index);
            jiv.setNCallee(nCallsTo);
            this.instanceVariables.set(index, jiv);
        }
    }
}

public void setGrade(double grade) {
    this.grade = grade;
}

public void setHasClassDoc(boolean hasDoc) {
    this.classDoc = hasDoc;
}

public void setClassName(String name) {
    this.className = name; // class name

    // see if the class name begins with a capital letter
    char c = name.charAt(0);
    if(Character.isUpperCase(c)) {
        this.hasCompliantName = true;
    } else {
        this.hasCompliantName = false;
    }
}

public void setMinimumGrade(double grade) {
    this.minimumGrade = grade;
}

public void setMinimumGradeMembers(Vector members) {
    this.minimumGradeMembers = members;
}

public void setNCommentLines(int number) {
    this.nCommentLines = number;
}

public void setNDisjointSets(int number) {
    this.nDisjointSets = number;
}

public void setNGlobalVariables(int number) {
    this.nGlobalVariables = number;
}

public void setNMethNamingPolicies(int number) {
    this.nMethNamingPolicies = number;
}

public void setNSourceLines(int number) {
    this.nSourceLines = number;
}

public void setNVarNamingPolicies(int number) {
    this.nVarNamingPolicies = number;
}

public void setPackageName(String name) {
    this.classPackage = name;
}
```

```
public void setRComments(double ratio) {
    this.rComments = ratio;
}

public void setRNotPrivateMembers (double ratio) {
    this.rNotPrivateMembers = ratio;
}

/* sorts all the methods by the number of calls to or from a method */
public void sortMethodCalls(int minCalls, int maxCalls) {
    int calls;
    JClassMethod jcm = null;
    JInstanceMethod jim = null;
    Vector allCalls = new Vector(maxCalls+1);
    //System.out.println("Made a vector of size " + (maxCalls+1));

    // make empty Vectors from 0 to maxCalls (the buckets)
    for (int i=0; i<maxCalls+1; i++) {
        allCalls.addElement(new Vector());
    }

    // class methods (bucket sort)
    for (int i=0; i<this.getNClassMethods(); i++) {
        calls = 0;
        jcm = this.getClassMethod(i);
        calls += jcm.getNCaller();
        calls += jcm.getNCallee();
        Vector oldV = (Vector) allCalls.elementAt(calls);
        oldV.addElement(jcm.getMethodName());
        allCalls.setElementAt(oldV, calls);
        //System.out.println("Added " + jcm.getMethodName() +
        //    " to allCalls(" + calls + ")");
    }

    // instance methods (bucket sort)
    for (int i=0; i<this.getNInstanceMethods(); i++) {
        calls = 0;
        jim = this.getInstanceMethod(i);
        calls += jim.getNCaller();
        calls += jim.getNCallee();
        Vector oldV = (Vector) allCalls.elementAt(calls);
        oldV.addElement(jim.getMethodName());
        allCalls.setElementAt(oldV, calls);
        //System.out.println("Added " + jim.getMethodName() +
        //    " to allCalls(" + calls + ")");
    }

    this.sortedMethodCalls = allCalls;
}

/* sorts all the variables by the number of calls made to them */
public void sortVariableCalls(int minCalls, int maxCalls) {
    int calls;
    JClassVariable jcv = null;
    JInstanceVariable jiv = null;
    Vector allCalls = new Vector(maxCalls+1);
    //System.out.println("Made a vector of size " + (maxCalls+1));

    // make empty Vectors from 0 to maxCalls (the buckets)
    for (int i=0; i<maxCalls+1; i++) {
        allCalls.addElement(new Vector());
    }

    // class variables (bucket sort)
    for (int i=0; i<this.getNClassVariables(); i++) {
        calls = 0;
        jcv = this.getClassVariable(i);
        calls += jcv.getNCallee();
        Vector oldV = (Vector) allCalls.elementAt(calls);
        oldV.addElement(jcv.getVariableName());
        allCalls.setElementAt(oldV, calls);
        //System.out.println("Added " + jcv.getVariableName() +
        //    " to allCalls(" + calls + ")");
    }

    // instance variables (bucket sort)
    for (int i=0; i<this.getNInstanceVariables(); i++) {
        calls = 0;
        jiv = this.getInstanceVariable(i);
        calls += jiv.getNCallee();
        Vector oldV = (Vector) allCalls.elementAt(calls);
```

```
            oldV.addElement(jiv.getVariableName());
            allCalls.setElementAt(oldV, calls);
            //System.out.println("Added " + jiv.getVariableName() +
            //    " to allCalls(" + calls + ")");
        }

        this.sortedVariableCalls = allCalls;
    }

    /* prints a class summary */
    public String toString() {
        return ("\nSummary for the class " + this.getClassName() + "\n" +
                "\tNumber of lines: " + this.getNSourceLines() + "\n" +
                "\tNumber of commented lines: " + this.getNCommentLines() + "\n" +
                "\tNumber of Instance variables: " + this.getNInstanceVariables() +
                "\n" +
                "\tNumber of Class variables: " + this.getNClassVariables() + "\n" +
                "\tNumber of Instance methods: " + this.getNInstanceMethods() +
                "\n" +
                "\tNumber of Class methods: " + this.getNClassMethods() + "\n" +
                "\tHas class documentation: " + this.hasClassDoc() + "\n" +
                "\tNumber of members: " + this.getNMembers() + "\n" +
                "\tNumber of public members: " + this.getNPublicMembers() + "\n" +
                "\tNumber of non-private members: " + this.getNNotPrivateMembers()
                + "\n" +
                "\tRatio of non-private members: " + this.getRNotPrivateMembers() +
                "\n" +
                "\tNumber of global variables: " + this.getNGlobalVariables() +
                "\n" +
                "\tRatio of comments: " + this.getRComments() + "\n" +
                "\tHas compliant class name: " + this.getHasCompliantName() + "\n" +
                "\tNumber of disjoint sets: " + this.getNDisjointSets() + "\n" +
                "\tNumber of method naming policies: " +
                this.getNMethNamingPolicies() + "\n" +
                "\tNumber of variable naming policies: " +
                this.getNVarNamingPolicies() + "\n" +
                "\tGrade: " + this.getGrade() + "\n" +
                "\tMinimum member grade: " + this.getMinimumGrade());
    }
}
```

JClassMethod.java

```
/**
 * JClassMethod
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JClassMethod extends JMethod {

    /* Instance Variables */
    private boolean isFinal = false; // whether its a constant

    /* Public Methods */

    /**
     * Class constructor
     */
    public JClassMethod() {}

    /**
     * Class constructor
     */
    public JClassMethod(String name, String visibility, boolean isFinal,
                        boolean foundDoc, String label) {
        this.methodName = name;
        char c = name.charAt(0);
        if (Character.isLowerCase(c)) {
            this.hasCompliantName = true;
        } else {
            this.hasCompliantName = false;
        }
        this.methodType = "class";
```

```
        this.visibility = visibility; // public, protected, private or default
        this.isFinal = isFinal; // if its final - true, else false
        this.hasMethodDoc = foundDoc; // method documentation header
        this.lineLengths = new Vector(100); // length of each line for 100 lines
        this.label = label;
    }

    public boolean getIsFinal() {
        return this.isFinal;
    }

    public String toString() {
        return("Summary for: " + this.getMethodName() + " (" + this.getLabel() +
               ")\n" +
               "\tNumber of lines: " + this.getNSourceLines() + "\n" +
               "\tNumber of commented lines: " + this.getNCommentLines() + "\n" +
               "\tMethod visibility: " + this.getVisibility() + "\n" +
               "\tHas method documentation: " + this.getHasMethodDoc() + "\n" +
               "\tIs declared final: " + this.getIsFinal() + "\n" +
               "\tNumber of calls made: " + this.getNCaller() + "\n" +
               "\tNumber of calls to this: " + this.getNCallee() + "\n" +
               "\tHas compliant name: " + this.getHasCompliantName() + "\n" +
               "\tAverage line length: " + this.getAvgLineLength() + "\n" +
               "\tMaximum line length: " + this.getMaxLineLength() + "\n" +
               "\tRatio of comments: " + this.getRComments() + "\n" +
               "\tGrade: " + this.getGrade());
    }
}
```

```
/**
 * JClassVariable
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

public class JClassVariable extends JVariable {

    /* Instance Variables */
    private boolean isFinal; // whether its a constant
    private boolean isGlobal; // if its public and not final

    /* Public Methods */

    /**
     * Class constructor
     */
    public JClassVariable(String name, String visibility, boolean isFinal,
                          boolean foundDoc, String label) {
        this.variableName = name;
        char c = name.charAt(0);
        if (!isFinal) {
            // should be initial lower case, rest title case
            if (Character.isLowerCase(c)) {
                this.hasCompliantName = true;
            } else {
                this.hasCompliantName = false;
            }
        } else {
            // constant - should have all upper case letter delimited by underscores
            boolean isCompliant = true;
            for (int i=0; i<name.length(); i++) {
                c = name.charAt(i);
                if (Character.isLetter(c) && Character.isLowerCase(c)) {
                    isCompliant = false;
                }
            }
            if (isCompliant) {
                this.hasCompliantName = true;
            } else {
                this.hasCompliantName = false;
            }
        }
        this.visibility = visibility;
        this.isFinal = isFinal;
        if (visibility.equals("public") && !isFinal) {
            isGlobal = true;
        } else {
```

```
            isGlobal = false;
        }
        this.variableDoc = foundDoc;
        this.variableType = "class";
        this.label = label;
    }

    public boolean getIsFinal() {
        return this.isFinal;
    }

    public boolean getIsGlobal() {
        return this.isGlobal;
    }

    public String toString() {
        return("Summary for: " + this.getVariableName() + " (" + this.getLabel() +
               ")\n" +
               "\tVariable visibility: " + this.getVisibility() + "\n" +
               "\tHas variable documentation: " + this.hasVariableDoc() + "\n" +
               "\tIs final: " + this.getIsFinal() + "\n" +
               "\tNumber of calls to this: " + this.getNCallee() + "\n" +
               "\tIs a global variable: " + this.getIsGlobal() + "\n" +
               "\tHas compliant name: " + this.getHasCompliantName() + "\n" +
               "\tGrade: " + this.getGrade());
    }
}
```

JColorConstants.java

```
/*
 *
 * JColorConstants
 *
 * Used by JVisualizer to color elements based on their grade
 *
 * Andrea Goethals
 *
 */

import java.awt.Color;
import javax.vecmath.*;

public class JColorConstants {

    /* class variables */
    // for grading elements
    public static Color dkGreen = new Color(0, 106, 25);
    public static Color dkGreen_green = new Color(0, 178, 0);
    public static Color green = new Color(0, 255, 0);
    public static Color green_greenYellow = new Color(113, 236, 6);
    public static Color green_yellow = new Color(180, 236, 11);
    public static Color greenYellow_yellow = new Color(213, 236, 6);
    public static Color yellow = new Color(255, 255, 0);
    public static Color yellow_yellowOrange = new Color(235, 206, 19);
    public static Color yellow_orange = new Color(236, 179, 7);
    public static Color yellowOrange_orange = new Color(236, 143, 13);
    public static Color orange = new Color(255, 126, 0);
    public static Color orange_orangeRed = new Color(255, 113, 24);
    public static Color orange_red = new Color(255, 101, 49);
    public static Color orangeRed_red = new Color(255, 0, 0);
    public static Color red = new Color(233, 0, 0);
    public static Color red_dkishred = new Color(210, 0, 0);
    public static Color dkishred = new Color(188, 0, 0);
    public static Color dkishred_dkred = new Color(165, 0, 0);
    public static Color dkred = new Color(143, 0, 0);
    public static Color dkred_verydkred = new Color(120, 0, 0);
    public static Color verydkred = new Color(98, 0, 0);

    // for reference elements
    private static Color black = new Color(0, 0, 0);
    private static Color blue = new Color(0, 0, 255);
    private static Color ltgray = new Color(204, 204, 204);
    private static Color white = new Color(255, 255, 255);

    // for grading elements
    public static final Color3f G100 = new Color3f(dkGreen);
```

```
    public static final Color3f G95 = new Color3f(dkGreen_green);
    public static final Color3f G90 = new Color3f(green);
    public static final Color3f G85 = new Color3f(green_greenYellow);
    public static final Color3f G80 = new Color3f(green_yellow);
    public static final Color3f G75 = new Color3f(greenYellow_yellow);
    public static final Color3f G70 = new Color3f(yellow);
    public static final Color3f G65 = new Color3f(yellow_yellowOrange);
    public static final Color3f G60 = new Color3f(yellow_orange);
    public static final Color3f G55 = new Color3f(yellowOrange_orange);
    public static final Color3f G50 = new Color3f(orange);
    public static final Color3f G45 = new Color3f(orange_orangeRed);
    public static final Color3f G40 = new Color3f(orange_red);
    public static final Color3f G35 = new Color3f(orangeRed_red);
    public static final Color3f G30 = new Color3f(red);
    public static final Color3f G25 = new Color3f(red_dkishred);
    public static final Color3f G20 = new Color3f(dkishred);
    public static final Color3f G15 = new Color3f(dkishred_dkred);
    public static final Color3f G10 = new Color3f(dkred);
    public static final Color3f G5 = new Color3f(dkred_verydkred);
    public static final Color3f G0 = new Color3f(verydkred);

    // for reference elements
    public static final Color3f BLACK = new Color3f(black);
    public static final Color3f LTGRAY = new Color3f(ltgray);
    public static final Color3f REF = new Color3f(blue);
    public static final Color GRAYBLUE = new Color(192, 206, 255);
    public static final Color GRAY = new Color(204, 204, 204);
    public static final Color3f WHITE = new Color3f(white);

}
```

JCritique.java

```
/**
 * JCritique
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.io.*;
import java.util.*;

public class JCritique {
    private Properties metrics;
    private Properties grading;
    private static final String CUSTOM_GRADING_FILE = "config/customGrading";
    private static final String CUSTOM_METRICS_FILE = "config/customMetrics";
    private static final String DEFAULT_GRADING_FILE = "config/defaultGrading";
    private static final String DEFAULT_METRICS_FILE = "config/defaultMetrics";

    public JCritique() {
        this.metrics = null;
        this.grading = null;
    }

    public void doCritique(JClass theClass) {
        System.out.println("\tBeginning to critique " + theClass.getClassName() +
                           "...");

        // import the default metrics and penalties
        this.importMetrics();
        this.importGrading();

        /*System.out.println("\nMetrics properties:");
        this.metrics.list(System.out);
        System.out.println("\nGrading properties:");
        this.grading.list(System.out);*/

        // grade the class
        this.gradeClass(theClass);

        // locate the worst offender(s)
        this.findLowestGrade(theClass);
    }

    public void findLowestGrade(JClass theClass) {
```

```
    JClassVariable jcv = null;
    JInstanceVariable jiv = null;
    JClassMethod jcm = null;
    JInstanceMethod jim = null;
    double minimumGrade = 100.0;
    Vector minimumGradeMembers = new Vector(10);

    // look through class variables
    for (int i=0; i<theClass.getNClassVariables(); i++) {
        jcv = theClass.getClassVariable(i);
        if (jcv.getGrade() < minimumGrade) {
            minimumGrade = jcv.getGrade();
            minimumGradeMembers = new Vector(10);
            minimumGradeMembers.addElement(jcv.getVariableName());
        } else if (jcv.getGrade() == minimumGrade) {
            minimumGradeMembers.addElement(jcv.getVariableName());
        }
    }
    // look through instance variables
    for (int i=0; i<theClass.getNInstanceVariables(); i++) {
        jiv = theClass.getInstanceVariable(i);
        if (jiv.getGrade() < minimumGrade) {
            minimumGrade = jiv.getGrade();
            minimumGradeMembers = new Vector(10);
            minimumGradeMembers.addElement(jiv.getVariableName());
        } else if (jiv.getGrade() == minimumGrade) {
            minimumGradeMembers.addElement(jiv.getVariableName());
        }
    }
    // look through class methods
    for (int i=0; i<theClass.getNClassMethods(); i++) {
        jcm = theClass.getClassMethod(i);
        if (jcm.getGrade() < minimumGrade) {
            minimumGrade = jcm.getGrade();
            minimumGradeMembers = new Vector(10);
            minimumGradeMembers.addElement(jcm.getMethodName());
        } else if (jcm.getGrade() == minimumGrade) {
            minimumGradeMembers.addElement(jcm.getMethodName());
        }
    }
    // look through instance methods
    for (int i=0; i<theClass.getNInstanceMethods(); i++) {
        jim = theClass.getInstanceMethod(i);
        if (jim.getGrade() < minimumGrade) {
            minimumGrade = jim.getGrade();
            minimumGradeMembers = new Vector(10);
            minimumGradeMembers.addElement(jim.getMethodName());
        } else if (jim.getGrade() == minimumGrade) {
            minimumGradeMembers.addElement(jim.getMethodName());
        }
    }

    theClass.setMinimumGrade(minimumGrade);
    theClass.setMinimumGradeMembers(minimumGradeMembers);

    /*System.out.print("\nLowest-quality members:\n\t");
    for (int i=0; i<minimumGradeMembers.size(); i++) {
        System.out.print(minimumGradeMembers.elementAt(i).toString() + " ");
    }*/
}

private void gradeClass(JClass theClass) {
    Vector criticism = null;
    JClassVariable jcv = null;
    JInstanceVariable jiv = null;
    JClassMethod jcm = null;
    JInstanceMethod jim = null;
    double grade = 100.0;

    /* critique variables */
    // class variables
    for (int i=0; i<theClass.getNClassVariables(); i++) {
        grade = 100.0;
        jcv = theClass.getClassVariable(i);
        // existence of class/instance variable documentation
        if (jcv.hasVariableDoc() &&
            this.metrics.getProperty("V_BOOL_HAS-DOC").equalsIgnoreCase("false")
                || !jcv.hasVariableDoc() &&
            this.metrics.getProperty("V_BOOL_HAS-DOC").equalsIgnoreCase("true")){
              grade +=
              Double.parseDouble(this.grading.getProperty("NOT_V_BOOL_HAS-DOC"));
```

```
                jcv.addCriticism("Existence of variable documentation = " +
                                 jcv.hasVariableDoc() + ": " +
                                 this.grading.getProperty("NOT_V_BOOL_HAS-DOC")
                                 + " points");
            }
            // compliant class/instance variable name
            if (jcv.getHasCompliantName() &&
                this.metrics.getProperty("V_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                                "false")
                || !jcv.getHasCompliantName() &&
                this.metrics.getProperty("V_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                                "true")){
                    grade +=
                    Double.parseDouble(this.grading.getProperty(
                                       "NOT_V_BOOL_IS-NAME-COMP"));
                    jcv.addCriticism("Compliant variable name = " +
                                 jcv.getHasCompliantName() + ": " +
                                 this.grading.getProperty(
                                     "NOT_V_BOOL_IS-NAME-COMP")
                                 + " points");
            }
            jcv.setGrade(grade);

            /*criticism = jcv.getCriticism();
            System.out.println("\nCriticism for " + jcv.getVariableName() + ": ");
            for (int j=0; j<criticism.size(); j++) {
                System.out.println("\t" + criticism.elementAt(j));
            }*/
        }
        // instance variables
        for (int i=0; i<theClass.getNInstanceVariables(); i++) {
            grade = 100.0;
            jiv = theClass.getInstanceVariable(i);
            // existence of class/instance variable documentation
            if (jiv.hasVariableDoc() &&
                this.metrics.getProperty("V_BOOL_HAS-DOC").equalsIgnoreCase("false")
                || !jiv.hasVariableDoc() &&
                this.metrics.getProperty("V_BOOL_HAS-DOC").equalsIgnoreCase("true")){
                  grade +=
                  Double.parseDouble(this.grading.getProperty("NOT_V_BOOL_HAS-DOC"));
                    jiv.addCriticism("Existence of variable documentation = " +
                                 jiv.hasVariableDoc() + ": " +
                                 this.grading.getProperty("NOT_V_BOOL_HAS-DOC")
                                 + " points");
            }
            // compliant class/instance variable name
            if (jiv.getHasCompliantName() &&
                this.metrics.getProperty("V_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                                 "false")
                || !jiv.getHasCompliantName() &&
                this.metrics.getProperty("V_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                                 "true")){
                    grade +=
                    Double.parseDouble(this.grading.getProperty(
                                         "NOT_V_BOOL_IS-NAME-COMP"));
                    jiv.addCriticism("Compliant variable name = " +
                                 jiv.getHasCompliantName() + ": " +
                                 this.grading.getProperty(
                                     "NOT_V_BOOL_IS-NAME-COMP")
                                 + " points");
            }
            jiv.setGrade(grade);

            /*criticism = jiv.getCriticism();
            System.out.println("\nCriticism for " + jiv.getVariableName() + ": ");
            for (int j=0; j<criticism.size(); j++) {
                System.out.println("\t" + criticism.elementAt(j));
            }*/

        }

        /* critique methods */
        // class methods
        for (int i=0; i<theClass.getNClassMethods(); i++) {
            grade = 100.0;
            jcm = theClass.getClassMethod(i);
            // maximum acceptable number of lines of code per method
            if (jcm.getNSourceLines() >
                Integer.parseInt(this.metrics.getProperty("M_INT_MAX-NUM-LOC"))) {
                    grade +=
                    Double.parseDouble(this.grading.getProperty(
```

```
                                "MORE_M_INT_MAX-NUM-LOC"));
            jcm.addCriticism("Lines of code = " + jcm.getNSourceLines() +
                             ": " +
                             this.grading.getProperty(
                                 "MORE_M_INT_MAX-NUM-LOC")
                             + " points");
    }
    // maximum acceptable maximum number of characters per line per method
    if (jcm.getMaxLineLength() >
        Integer.parseInt(this.metrics.getProperty(
                                    "M_INT_MAX-MAX-LINE-LENGTH"))) {
            grade +=
            Double.parseDouble(this.grading.getProperty(
                               "MORE_M_INT_MAX-MAX-LINE-LENGTH"));
            jcm.addCriticism("Maximum line length = " +
                             jcm.getMaxLineLength() + ": " +
                             this.grading.getProperty(
                                 "MORE_M_INT_MAX-MAX-LINE-LENGTH")
                             + " points");
    }
    // maximum acceptable average number of characters per line per method
    if (jcm.getAvgLineLength() >
        Integer.parseInt(this.metrics.getProperty(
                             "M_INT_MAX-AVG-LINE-LENGTH"))) {
            grade +=
            Double.parseDouble(this.grading.getProperty(
                               "MORE_M_INT_MAX-AVG-LINE-LENGTH"));
            jcm.addCriticism("Average line length = " +
                             jcm.getAvgLineLength() + ": " +
                             this.grading.getProperty(
                                 "MORE_M_INT_MAX-AVG-LINE-LENGTH")
                             + " points");
    }
    // minimum acceptable ratio of method comment density
    if (jcm.getRComments() <
        Double.parseDouble(this.metrics.getProperty(
                               "M_FLOAT_MIN-RAT-COM-DENSITY"))){
            grade +=
            Double.parseDouble(this.grading.getProperty(
                               "LESS_M_FLOAT_MIN-RAT-COM-DENSITY"));
            jcm.addCriticism("Comment density = " + jcm.getRComments() +
                             ": " +
                             this.grading.getProperty(
                                 "LESS_M_FLOAT_MIN-RAT-COM-DENSITY")
                             + " points");
    }
    // existence of class/instance method documentation
    if (jcm.getHasMethodDoc() &&
        this.metrics.getProperty("M_BOOL_HAS-DOC").equalsIgnoreCase("false")
        || !jcm.getHasMethodDoc() &&
        this.metrics.getProperty("M_BOOL_HAS-DOC").equalsIgnoreCase("true")){
            grade +=
            Double.parseDouble(this.grading.getProperty(
                                 "NOT_M_BOOL_HAS-DOC"));
            jcm.addCriticism("Existence of method documentation = " +
                             jcm.getHasMethodDoc() + ": " +
                             this.grading.getProperty("NOT_M_BOOL_HAS-DOC")
                             + " points");
    }
    // compliant class/instance method name
    if (jcm.getHasCompliantName() &&
        this.metrics.getProperty("M_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                        "false")
        || !jcm.getHasCompliantName() &&
        this.metrics.getProperty("M_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                        "true")){
            grade +=
            Double.parseDouble(this.grading.getProperty(
                                 "NOT_M_BOOL_IS-NAME-COMP"));
            jcm.addCriticism("Compliant method name = " +
                             jcm.getHasCompliantName() + ": " +
                             this.grading.getProperty(
                                 "NOT_M_BOOL_IS-NAME-COMP")
                             + " points");
    }
    jcm.setGrade(grade);

    /*criticism = jcm.getCriticism();
    System.out.println("\nCriticism for " + jcm.getMethodName() + ": ");
    for (int j=0; j<criticism.size(); j++) {
        System.out.println("\t" + criticism.elementAt(j));
```

```
    }*/
}

// instance methods
for (int i=0; i<theClass.getNInstanceMethods(); i++) {
    grade = 100.0;
    jim = theClass.getInstanceMethod(i);
    // maximum acceptable number of lines of code per method
    if (jim.getNSourceLines() >
        Integer.parseInt(this.metrics.getProperty("M_INT_MAX-NUM-LOC"))) {
            grade +=
            Double.parseDouble(this.grading.getProperty(
                                "MORE_M_INT_MAX-NUM-LOC"));
            jim.addCriticism("Lines of code = " + jim.getNSourceLines() +
                             ": " +
                             this.grading.getProperty(
                                 "MORE_M_INT_MAX-NUM-LOC")
                             + " points");
    }
    // maximum acceptable maximum number of characters per line per method
    if (jim.getMaxLineLength() >
        Integer.parseInt(this.metrics.getProperty(
                             "M_INT_MAX-MAX-LINE-LENGTH"))){
            grade +=
            Double.parseDouble(this.grading.getProperty(
                                "MORE_M_INT_MAX-MAX-LINE-LENGTH"));
            jim.addCriticism("Maximum line length = " +
                             jim.getMaxLineLength() + ": " +
                             this.grading.getProperty(
                                 "MORE_M_INT_MAX-MAX-LINE-LENGTH")
                             + " points");
    }
    // maximum acceptable average number of characters per line per method
    if (jim.getAvgLineLength() >
        Integer.parseInt(this.metrics.getProperty(
                             "M_INT_MAX-AVG-LINE-LENGTH"))){
            grade +=
            Double.parseDouble(this.grading.getProperty(
                               "MORE_M_INT_MAX-AVG-LINE-LENGTH"));
            jim.addCriticism("Average line length = " +
                             jim.getAvgLineLength() + ": " +
                             this.grading.getProperty(
                                 "MORE_M_INT_MAX-AVG-LINE-LENGTH")
                             + " points");
    }
    // minimum acceptable ratio of method comment density
    if (jim.getRComments() <
        Double.parseDouble(this.metrics.getProperty(
                               "M_FLOAT_MIN-RAT-COM-DENSITY"))){
            grade +=
            Double.parseDouble(this.grading.getProperty(
                                "LESS_M_FLOAT_MIN-RAT-COM-DENSITY"));
            jim.addCriticism("Comment density = " + jim.getRComments() +
                             ": " +
                             this.grading.getProperty(
                                 "LESS_M_FLOAT_MIN-RAT-COM-DENSITY")
                             + " points");
    }
    // existence of class/instance method documentation
    if (jim.getHasMethodDoc() &&
        this.metrics.getProperty("M_BOOL_HAS-DOC").equalsIgnoreCase("false")
        || !jim.getHasMethodDoc() &&
        this.metrics.getProperty("M_BOOL_HAS-DOC").equalsIgnoreCase(
                                                   "true")) {
            grade +=
            Double.parseDouble(this.grading.getProperty(
                                "NOT_M_BOOL_HAS-DOC"));
            jim.addCriticism("Existence of method documentation = " +
                             jim.getHasMethodDoc() + ": " +
                             this.grading.getProperty("NOT_M_BOOL_HAS-DOC")
                             + " points");
    }
    // compliant class/instance method name
    if (jim.getHasCompliantName() &&
        this.metrics.getProperty("M_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                        "false")
        || !jim.getHasCompliantName() &&
        this.metrics.getProperty("M_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                        "true")){
            grade +=
            Double.parseDouble(this.grading.getProperty(
```

```
                                    "NOT_M_BOOL_IS-NAME-COMP"));
            jim.addCriticism("Compliant method name = " +
                             jim.getHasCompliantName() + ": " +
                             this.grading.getProperty(
                                 "NOT_M_BOOL_IS-NAME-COMP")
                             + " points");
    }
    jim.setGrade(grade);

    /*criticism = jim.getCriticism();
    System.out.println("\nCriticism for " + jim.getMethodName() + ": ");
    for (int j=0; j<criticism.size(); j++) {
        System.out.println("\t" + criticism.elementAt(j));
    }*/
}

/* critique class */
grade = 100.0;
// maximum acceptable number of disjoint sets
if (theClass.getNDisjointSets() >
    Integer.parseInt(this.metrics.getProperty(
                        "C_INT_MAX-NUM-DISJOINT-SETS"))){
        grade +=
        Double.parseDouble(this.grading.getProperty(
                            "MORE_C_INT_MAX-NUM-DISJOINT-SETS"));
        theClass.addCriticism("Number of cohesive member sets = " +
                            theClass.getNDisjointSets() + ": " +
                            this.grading.getProperty(
                                "MORE_C_INT_MAX-NUM-DISJOINT-SETS")
                            + " points");
}
// maximum acceptable number of public class/instance variables
// and methods
if (theClass.getNPublicMembers() >
    Integer.parseInt(this.metrics.getProperty("C_INT_MAX-NUM-PUB-MEMS"))) {
        grade +=
        Double.parseDouble(this.grading.getProperty(
                            "MORE_C_INT_MAX-NUM-PUB-MEMS"));
        theClass.addCriticism("Number of public members = " +
                            theClass.getNPublicMembers() + ": " +
                            this.grading.getProperty(
                                "MORE_C_INT_MAX-NUM-PUB-MEMS")
                            + " points");
}
// maximum acceptable ratio of public, protected and/or
// default class/instance variables and methods to all
// class/instance variables and methods
if (theClass.getRNotPrivateMembers() >
    Double.parseDouble(this.metrics.getProperty(
                        "C_FLOAT_MAX-RAT-NONPRIV-MEMS"))){
        grade +=
        Double.parseDouble(this.grading.getProperty(
                            "MORE_C_FLOAT_MAX-RAT-NONPRIV-MEMS"));
        theClass.addCriticism("Ratio of non-private members = " +
                            theClass.getRNotPrivateMembers() + ": " +
                            this.grading.getProperty(
                                "MORE_C_FLOAT_MAX-RAT-NONPRIV-MEMS")
                            + " points");
}
// maximum acceptable number of public class variables that
// are not declared final
if (theClass.getNGlobalVariables() >
    Integer.parseInt(this.metrics.getProperty(
                        "C_INT_MAX-NUM-GLOBAL-VARS"))) {
        grade +=
        Double.parseDouble(this.grading.getProperty(
                            "MORE_C_INT_MAX-NUM-GLOBAL-VARS"));
        theClass.addCriticism("Number of global variables = " +
                            theClass.getNGlobalVariables() + ": " +
                            this.grading.getProperty(
                                "MORE_C_INT_MAX-NUM-GLOBAL-VARS")
                            + " points");
}
// maximum acceptable number of lines of code per class
if (theClass.getNSourceLines() >
    Integer.parseInt(this.metrics.getProperty("C_INT_MAX-NUM-LOC"))) {
        grade +=
        Double.parseDouble(this.grading.getProperty(
                            "MORE_C_INT_MAX-NUM-LOC"));
        theClass.addCriticism("Lines of code = " + theClass.getNSourceLines()
                            + ": " +
```

```
                                this.grading.getProperty(
                                    "MORE_C_INT_MAX-NUM-LOC")
                                + " points");
        }
        // maximum acceptable number of class/instance method naming policies
        if (theClass.getNMethNamingPolicies() >
            Integer.parseInt(this.metrics.getProperty(
                                "C_INT_MAX-NUM-METH-NAME-POLS"))){
                grade +=
                Double.parseDouble(this.grading.getProperty(
                                 "MORE_C_INT_MAX-NUM-METH-NAME-POLS"));
                theClass.addCriticism("Number of method-naming policies = " +
                                theClass.getNMethNamingPolicies() + ": " +
                                this.grading.getProperty(
                                    "MORE_C_INT_MAX-NUM-METH-NAME-POLS")
                                + " points");
        }
        // maximum acceptable number of class/instance variable naming policies
        if (theClass.getNVarNamingPolicies() >
            Integer.parseInt(this.metrics.getProperty(
                                "C_INT_MAX-NUM-VAR-NAME-POLS"))){
                grade +=
                Double.parseDouble(this.grading.getProperty(
                                  "MORE_C_INT_MAX-NUM-VAR-NAME-POLS"));
                theClass.addCriticism("Number of variable-naming policies = " +
                                theClass.getNVarNamingPolicies() + ": " +
                                this.grading.getProperty(
                                    "MORE_C_INT_MAX-NUM-VAR-NAME-POLS")
                                + " points");
        }
        // minimum acceptable ratio of class comment density
        if (theClass.getRComments() <
            Double.parseDouble(this.metrics.getProperty(
                                "C_FLOAT_MIN-RAT-COM-DENSITY"))){
                grade +=
                Double.parseDouble(this.grading.getProperty(
                                  "LESS_C_FLOAT_MIN-RAT-COM-DENSITY"));
                theClass.addCriticism("Comment density = " + theClass.getRComments()
                                + ": " +
                                this.grading.getProperty(
                                    "LESS_C_FLOAT_MIN-RAT-COM-DENSITY")
                                + " points");
        }
        // existence of class documentation
        if (theClass.hasClassDoc() &&
            this.metrics.getProperty("C_BOOL_HAS-DOC").equalsIgnoreCase("false") ||
            !theClass.hasClassDoc() &&
            this.metrics.getProperty("C_BOOL_HAS-DOC").equalsIgnoreCase("true")) {
                grade +=
                Double.parseDouble(this.grading.getProperty("NOT_C_BOOL_HAS-DOC"));
                theClass.addCriticism("Existence of class documentation = " +
                                theClass.hasClassDoc() + ": " +
                                this.grading.getProperty("NOT_C_BOOL_HAS-DOC")
                                + " points");
        }
        // compliant class name
        if (theClass.getHasCompliantName() &&
            this.metrics.getProperty("C_BOOL_IS-NAME-COMP").equalsIgnoreCase("false")
            || !theClass.getHasCompliantName() &&
            this.metrics.getProperty("C_BOOL_IS-NAME-COMP").equalsIgnoreCase(
                                                        "true")){
                grade +=
                Double.parseDouble(this.grading.getProperty(
                                "NOT_C_BOOL_IS-NAME-COMP"));
                theClass.addCriticism("Compliant class name = " +
                                theClass.getHasCompliantName() + ": " +
                                this.grading.getProperty(
                                    "NOT_C_BOOL_IS-NAME-COMP")
                                + " points");
        }

        theClass.setGrade(grade);

        /*criticism = theClass.getCriticism();
        System.out.println("\nCriticism for " + theClass.getClassName() + ":");
        for (int j=0; j<criticism.size(); j++) {
             System.out.println("\t" + criticism.elementAt(j));
        }*/
    }

    private void importGrading() {
```

```
        // import the default penalties
        Properties defaultGrading = new Properties();
        try {
            FileInputStream defaultStream =
                new FileInputStream(DEFAULT_GRADING_FILE);
            defaultGrading.load(defaultStream);
            defaultStream.close();
        } catch (Exception e) {
            System.out.println("Can't load " + DEFAULT_GRADING_FILE);
        }

        // import the custom penalties
        this.grading = new Properties(defaultGrading);
        try {
            FileInputStream customStream = new FileInputStream(CUSTOM_GRADING_FILE);
            this.grading.load(customStream);
            customStream.close();
        } catch (Exception e) {
            System.out.println("Can't load " + CUSTOM_GRADING_FILE);
        }
    }

    private void importMetrics() {
        // import the default metrics
        Properties defaultMetrics = new Properties();
        try {
            FileInputStream defaultStream =
                new FileInputStream(DEFAULT_METRICS_FILE);
            defaultMetrics.load(defaultStream);
            defaultStream.close();
        } catch (Exception e) {
            System.out.println("Can't load " + DEFAULT_METRICS_FILE);
        }

        // import the custom metrics
        this.metrics = new Properties(defaultMetrics);
        try {
            FileInputStream customStream = new FileInputStream(CUSTOM_METRICS_FILE);
            this.metrics.load(customStream);
            customStream.close();
        } catch (Exception e) {
            System.out.println("Can't load " + CUSTOM_METRICS_FILE);
        }
    }
}
```

JGrammar.java

```
/**
 * JGrammar
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

public class JGrammar {

    public boolean isJDKClass(String theWord) {
        boolean result = false;
        if (theWord.equals("String") || theWord.equals("Vector") ||
            theWord.equals("Object") || theWord.equals("Integer") ||
            theWord.equals("StringTokenizer") || theWord.equals("Stack")) {
                result = true;
        }

        return result;
    }

    public boolean isKeyword(String theWord) {
        String word = theWord.toLowerCase();
        boolean result = false;
        if (word.equals("catch") || word.equals("class") ||
            word.equals("implements") || word.equals("extends") ||
            word.equals("throws") || word.equals("new") ||
            word.equals("import") || word.equals("package") ||
            word.equals("super") || word.equals("interface") ||
            word.equals("void") || word.equals("if") ||
```

```
        word.equals("else") || word.equals("for") ||
        word.equals("while") || word.equals("do") ||
        word.equals("try") || word.equals("switch") ||
        word.equals("finally") || word.equals("synchronized") ||
        word.equals("return") || word.equals("throw") ||
        word.equals("break") || word.equals("continue") ||
        word.equals("then") || word.equals("case") ||
        word.equals("default")) {
            result = true;
    }

    return result;
}

/**
 * Determines if the word is a modifier
 *
 * @param         theWord    A source code word
 * @return        boolean    true if its a modifier, else false
 */
public boolean isModifier(String theWord) {
    String word = theWord.toLowerCase();
    boolean result = false;

    if (word.equals("public") || word.equals("protected") ||
        word.equals("static") || word.equals("abstract") ||
        word.equals("final")  || word.equals("native") ||
       word.equals("synchronized") || word.equals("transient") ||
        word.equals("volatile") || word.equals("strictfp") ||
        word.equals("private")) {
            result = true;
    }

    return result;
}

public boolean isNumber(String theWord) {
    boolean result = false;

    if (theWord.startsWith("0") || theWord.startsWith("1") ||
        theWord.startsWith("2") || theWord.startsWith("3") ||
        theWord.startsWith("4") || theWord.startsWith("5") ||
        theWord.startsWith("6") || theWord.startsWith("7") ||
        theWord.startsWith("8") || theWord.startsWith("9")) {
            result = true;
    }
    return result;
}

public boolean isOperator(String theWord) {
    String word = theWord.toLowerCase();
    boolean result = false;

    if (word.equals("&&") || word.equals("||") || word.equals("|") ||
        word.equals("^") || word.equals("&") || word.equals("==") ||
        word.equals("!=") || word.equals("<") || word.equals(">") ||
        word.equals("<=") || word.equals(">=") || word.equals("<<") ||
        word.equals(">>") || word.equals(">>>") || word.equals("+") ||
        word.equals("-") || word.equals("/") || word.equals("%") ||
        word.equals("=") || word.equals("?") || word.equals(":")) {
            result = true;
    }
    return result;
}

/**
 * Determines if the word is a type
 *
 * @param         theWord    A source code word
 * @return        boolean    true if its a type, else false
 */
public boolean isType (String theWord) {
    String word = theWord.toLowerCase();
    boolean result = false;

    if (word.equals("byte") || word.equals("short") ||
        word.equals("char") || word.equals("int") ||
        word.equals("long")  || word.equals("float") ||
       word.equals("double") || word.equals("boolean")) {
            result = true;
    }
```

```
        return result;
    }

}
```

JInstanceMethod.java

```
/**
 * JInstanceMethod
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JInstanceMethod extends JMethod {
    /* Instance Variables */
    boolean isConstructor;

    /* Public Methods */

    /**
     * Class constructor
     */
    public JInstanceMethod(String name, String visibility, boolean foundDoc,
                           boolean isConstructor, String label) {
        this.methodName = name;
        char c = name.charAt(0);
        if (!isConstructor && Character.isLowerCase(c) ||
             isConstructor && Character.isUpperCase(c)) {
            this.hasCompliantName = true;
        } else {
            this.hasCompliantName = false;
        }
        this.methodType = "instance";
        this.visibility = visibility; // public, protected, private or default
        this.hasMethodDoc = foundDoc; // method documentation header
        this.lineLengths = new Vector(100); // length of each line for 100 lines
        this.isConstructor = isConstructor;
        this.label = label;
    }

    public boolean getIsConstructor() {
        return isConstructor;
    }

    public String toString() {
        return("Summary for: " + this.getMethodName() + " (" + this.getLabel() +
               ")\n" +
               "\tNumber of lines: " + this.getNSourceLines() + "\n" +
               "\tNumber of commented lines: " + this.getNCommentLines() + "\n" +
               "\tMethod visibility: " + this.getVisibility() + "\n" +
               "\tHas method documentation: " + this.getHasMethodDoc() + "\n" +
               "\tIs a constructor: " + this.getIsConstructor() + "\n" +
               "\tNumber of calls made: " + this.getNCaller() + "\n" +
               "\tNumber of calls to this: " + this.getNCallee() + "\n" +
               "\tHas compliant name: " + this.getHasCompliantName() + "\n" +
               "\tAverage line length: " + this.getAvgLineLength() + "\n" +
               "\tMaximum line length: " + this.getMaxLineLength() + "\n" +
               "\tRatio of comments: " + this.getRComments() + "\n" +
               "\tGrade: " + this.getGrade());
    }
}
```

JInstanceVariable.java

```
/**
 * JInstanceVariable
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */
```

```
public class JInstanceVariable extends JVariable {

    /* Public Methods */

    /**
     * Class constructor
     */
    public JInstanceVariable(String name, String visibility, boolean foundDoc,
                             String label) {
        this.variableName = name;
        char c = name.charAt(0);
        if (Character.isLowerCase(c)) {
            this.hasCompliantName = true;
        } else {
            this.hasCompliantName = false;
        }
        this.visibility = visibility;
        this.variableDoc = foundDoc;
        this.variableType = "instance";
        this.label = label;
    }
}
```

JListener.java

```
/*
 *
 * JListener
 *
 * Used by JVisualizer to react to key presses
 *
 * Andrea Goethals
 *
 */

import java.awt.*;
import java.awt.event.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import java.util.*;

public class JListener extends Behavior implements ItemListener {

    private TransformGroup transformGroup;
    private WakeupCriterion criterion;
    private String vScopeSelection;
    private static final String IN_CLASS = "Within the class";
    private static final String OUT_PACKAGE = "Another class in same package";
    private static final String OUT_PUBLIC = "Any other class";
    private static final String OUT_SUBCLASS = "A subclass";

    public JListener(TransformGroup tg) {
        this.transformGroup = tg;
    }

    public void initialize() {
        this.criterion = new WakeupOnBehaviorPost(this, ItemEvent.SELECTED);
        wakeupOn(criterion);
        //System.out.println("I initialized");
    }

    public void processStimulus(Enumeration criteria) {
        //System.out.println("Heard that selection!");
        if (vScopeSelection.equals(IN_CLASS)) {
            // make class transparent
            JModeler.boxAppearance.setTransparencyAttributes(JModeler.transparent);
            JModeler.defAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.privAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.protAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.pubAttributes.setColor(JColorConstants.LTGRAY);
        } else if (vScopeSelection.equals(OUT_PUBLIC)) {
            // make class opaque
            JModeler.boxAppearance.setTransparencyAttributes(JModeler.opaque);
            JModeler.defAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.privAttributes.setColor(JColorConstants.BLACK);
            JModeler.protAttributes.setColor(JColorConstants.BLACK);
```

```
            JModeler.pubAttributes.setColor(JColorConstants.LTGRAY);
        } else if (vScopeSelection.equals(OUT_PACKAGE)) {
            JModeler.boxAppearance.setTransparencyAttributes(JModeler.opaque);
            JModeler.defAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.privAttributes.setColor(JColorConstants.BLACK);
            JModeler.protAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.pubAttributes.setColor(JColorConstants.LTGRAY);
        } else if (vScopeSelection.equals(OUT_SUBCLASS)) {
            JModeler.boxAppearance.setTransparencyAttributes(JModeler.opaque);
            JModeler.defAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.privAttributes.setColor(JColorConstants.BLACK);
            JModeler.protAttributes.setColor(JColorConstants.LTGRAY);
            JModeler.pubAttributes.setColor(JColorConstants.LTGRAY);
        }
        wakeupOn(criterion);
    }

    public void itemStateChanged(ItemEvent e) {
        this.vScopeSelection = e.getItem().toString();
        postId(ItemEvent.SELECTED);
    }
}
```

JMethod.java

```
/**
 * JMethod
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JMethod {
    /* Instance Variables */
    protected int avgLineLength; // average line length of the method
    protected Vector criticism; // the guidelines broken by the method as Strings
    protected double grade; // the overall grade given to the method
    protected boolean hasCompliantName; // if name is compliant
    protected boolean hasMethodDoc; // if it has a method documentation heading
    protected String label; // "CM#" or "IM#", # is from 1 to n
    protected Vector lineLengths; // length of each line in method
    protected int maxLineLength;
    protected int nCallee;
    protected int nCaller;
    protected int nCommentLines; // the number of source lines in method scope
    protected int nSourceLines; // the number of source lines in method scope
    protected String methodName; // the method name
    protected String methodType; // "class", "instance", or null
    protected double rComments; // comment density
    protected Vector sourceLines; // all source lines in method scope
    protected String visibility; // visibility either: public,protected,private,default

    /* Public Methods */

    /**
     * Class constructor
     */
    public JMethod() {
        this.avgLineLength = 0;
        this.criticism = new Vector(10);
        this.grade = 100.0;
        this.hasCompliantName = false;
        this.hasMethodDoc = false;
        this.label = null;
        this.lineLengths = null;
        this.maxLineLength = 0;
        this.nCallee = 0;
        this.nCaller = 0;
        this.nCommentLines = 0;
        this.nSourceLines = 0;
        this.methodName = null;
        this.methodType = null;
        this.rComments =  0.0;
        this.sourceLines = new Vector(50);
        this.visibility = "default";
```

```
}

public void addCriticism(String criticism) {
    this.criticism.addElement(criticism);
}

public void addSourceLine(String line) {
    String nChars = "-1";
    nChars = String.valueOf(line.length());

    this.sourceLines.addElement(line);
    //System.out.println("Added line: " + line);

    this.lineLengths.addElement(nChars);
   //System.out.println("Add line length: " + nChars);
}

public int getAvgLineLength() {
    return this.avgLineLength;
}

public Vector getCriticism() {
    return this.criticism;
}

public double getGrade() {
    return this.grade;
}

public boolean getHasCompliantName() {
    return this.hasCompliantName;
}

public boolean getHasMethodDoc() {
    return hasMethodDoc;
}

public String getLabel() {
    return this.label;
}

public Vector getLineLengths() {
    return this.lineLengths;
}

public int getMaxLineLength() {
    return this.maxLineLength;
}

public String getMethodName() {
    return this.methodName;
}

public String getMethodType() {
    return this.methodType;
}

public int getNCallee() {
    return this.nCallee;
}

public int getNCaller() {
    return this.nCaller;
}

public int getNCommentLines() {
    return this.nCommentLines;
}

public int getNSourceLines() {
    return this.nSourceLines;
}

public double getRComments() {
    return this.rComments;
}

public String getSourceLine(int index) {
    return (String) this.sourceLines.elementAt(index);
}
```

```
    public String getVisibility() {
        return this.visibility;
    }

    public void incrementNCommentLines() {
        this.nCommentLines++;
    }

    public void incrementNSourceLines() {
        this.nSourceLines++;
    }

    public void setAvgLineLength(int number) {
        this.avgLineLength = number;
    }

    public void setGrade(double grade) {
        this.grade = grade;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public void setMaxLineLength(int number) {
        this.maxLineLength = number;
    }

    public void setMethodName(String name) {
        this.methodName = name;
    }

    public void setNCallee(int nCallsTo) {
        this.nCallee = nCallsTo;
    }

    public void setNCaller(int nCallsFrom) {
        this.nCaller = nCallsFrom;
    }

    public void setRComments(double number) {
        this.rComments = number;
    }

    public String toString() {
        return("Summary for: " + this.getMethodName() + "\n" +
               "\tNumber of lines: " + this.getNSourceLines() + "\n" +
               "\tNumber of commented lines: " + this.getNCommentLines() + "\n" +
               "\tMethod visibility: " + this.getVisibility() + "\n" +
               "\tHas method documentation: " + this.getHasMethodDoc());
    }
}
```

JModeler.java

```
/*
 *
 * JModeler
 *
 * Used by JVisualizer
 *
 * Andrea Goethals
 *
 */

import java.applet.Applet;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import javax.swing.*;
import java.util.Enumeration;
import com.sun.j3d.utils.applet.MainFrame;
import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.Box;
import com.sun.j3d.utils.geometry.Cylinder;
import com.sun.j3d.utils.geometry.Text2D;
```

```
import com.sun.j3d.utils.behaviors.mouse.*;
import com.sun.j3d.utils.behaviors.picking.*;
import com.sun.j3d.utils.picking.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import JColorConstants;
import JListener;

public class JModeler extends Applet {

    /* Instance variables */
    public static Appearance boxAppearance; // for the class
    private double classHeight; // height of the class box
    public static float classTransparency;
    private double classWidth; // width of the class box
    public static ColoringAttributes defAttributes; // for label extenders
    private int maxVariableCall;
    private static float methArea; // physical area in height where methods are drawn
    private int [] methCohDiff; // difference in cohesion value between the previous
                                // sorted method cohesion value and this one
    private float methCohScaleFactor;
    private Point3f [] methCoordsBottom; // coords for each meth to use as endpoints
                                         // to visualize a method use
    private Point3f [] methCoordsTop; // coords for each method to use as an endpoint
                                      // to visualize a method call
    private Vector methCoordsNames;
    private int minVariableCall;
    public static TransparencyAttributes opaque =
        new TransparencyAttributes(TransparencyAttributes.BLENDED, 0.0f); // class
    public static ColoringAttributes privAttributes; // for label extenders
    public static ColoringAttributes protAttributes; // for label extenders
    public static ColoringAttributes pubAttributes; // for label extenders
    private JCallMatrix theCallMatrix; // method calls data structure for the
                                       // class being visualized
    private JClass theClass; // the class being visualized
    public static TransparencyAttributes transparent =
        new TransparencyAttributes(TransparencyAttributes.BLENDED, 1.0f); // class
    private SimpleUniverse universe = null;
    private static float varArea; // physical area in height where the vars are drawn
    private int [] varCohDiff; // difference in cohesion value between the previous
                               // sorted var cohesion value and this one
    private float varCohScaleFactor;
    private Point3f [] varCoords; // coordinates for each var to use as an endpoint
                                  // to visualize a var use
    private Vector varCoordsNames;

    /* constants */
    private static final String INITIAL_STRING = "Welcome to JVisualizer\n\n" +
                "Navigation:\thold right mouse button to transpose\n" +
                "           \t\thold left mouse button to rotate\n" +
                "           \t\thold left mouse button FIRST then \"Alt\" to zoom";
    private static final float BOTTOM_BUFFER = 0.015f; // spacing for laying out
                                                       // methods and variables
    private static final float CALL_COORD_Z = -0.015f; // just back from the origin
                                                       // (and tucked into the
                                                       // box so that it is hidden
                                                       // inside the class)
    private static final float EXT_LENGTH = 0.15f; // label extension length
    private static final String IN_CLASS = "Within the class"; // text for the
                                                       // current view pull-down menu
    private static final float LEFT_BUFFER = 0.015f; // spacing for laying out
                                                     // methods and variables
    private static final int MAX_COHESION_DIFFERENCE = 50;
    private static final float MAX_LINE_WIDTH = 7.0f; // maximum line width to use
                                                      // for method calls
    private static final float MEMBER_BUFFER = 0.015f; // spacing for laying out
                                                       // methods and variables
    private static final float MEMBER_HEIGHT = 0.06f; // height of a method or var
    private static final float MEMBER_WIDTH = 0.06f; // width of a method or variable
    private static final float MIDDLE_BUFFER = 0.2f; // spacing for laying out
                                                     // methods and variables
    private static final float MIN_LINE_WIDTH = 1.0f; // minimum line width to use
                                                      // for method calls
    private static final String OUT_PACKAGE = "Another class in same package";
                                        // text for the current view pull-down menu
    private static final String OUT_PUBLIC = "Any other class";
                                        // text for the current view pull-down menu
    private static final String OUT_SUBCLASS = "A subclass";
                                        // text for the current view pull-down menu
    private static final float RIGHT_BUFFER = 0.015f; // spacing for laying out
                                                      // methods and variables
```

```
private static final float TOP_BUFFER = 0.015f; // spacing for laying out
                                                // methods and variables

/* GUI components */
private static Choice vScope = new Choice(); // current view pull-down menu
private static Label scopeLabel; // label to left of current view pull-down menu
private static Panel backPanel; // the entire GUI panel
private static Panel centralPanel; // the GUI panel behind 3D view and report
private static Panel northPanel; // the GUI panel behind the viewing scope menu
private static Panel southPanel; // the GUI panel behind the instructions and
                                 // color panel
private static TextArea tArea; // the instructions area
private static TextArea reportArea; // the report are
public static String textMessage = INITIAL_STRING; // the instructions text
private static String vScopeString = OUT_PUBLIC; // still used???

/* constructor
 */
public JModeler(JClass theClass, JCallMatrix theCallMatrix) {
    this.theClass = theClass;
    this.theCallMatrix = theCallMatrix;
}

/* for each var and method in class this calculates the single coordinate to use
 * to draw the call lines to or from the member
 */
private void calculateCallEndpoints() {
    float theX, theY, theYTop, theYBottom, theZ;
    float nDrawn;
    // the following 2 casts are used to get them to floats (eventually) so that
    // the compiler does not complain about a loss of precision
    Double classWidthDouble = new Double(classWidth);
    Double classHeightDouble = new Double(classHeight);

    // calculate method coordinates
    int nMethods = theClass.getNClassMethods() + theClass.getNInstanceMethods();
    this.methCoordsBottom = new Point3f[nMethods];
    this.methCoordsTop = new Point3f[nMethods];
    for (int i=0; i<nMethods; i++) {
        nDrawn = (float) i; // number "drawn"
        theX = 0.0f - classWidthDouble.floatValue() + LEFT_BUFFER +
               MEMBER_WIDTH +
              (nDrawn * 2.0f * MEMBER_WIDTH) + (nDrawn * MEMBER_BUFFER);
        theYBottom = 0.0f + classHeightDouble.floatValue() - TOP_BUFFER
                     - varArea -
                     MIDDLE_BUFFER -
                     (methCohScaleFactor * methCohDiff[i] * MEMBER_HEIGHT) -
                     MEMBER_HEIGHT;
        theYTop = 0.0f + classHeightDouble.floatValue() - TOP_BUFFER - varArea -
                  MIDDLE_BUFFER -
                  (methCohScaleFactor * methCohDiff[i] * MEMBER_HEIGHT) +
                  MEMBER_HEIGHT;
        theZ = CALL_COORD_Z;
        methCoordsBottom[i] = new Point3f(theX, theYBottom, theZ);
        methCoordsTop[i] = new Point3f(theX, theYTop, theZ);
        //System.out.println("Meth X: " + theX + " YTop: " + theYTop +
        //    " YBot: " + theYBottom + " Z: " + theZ);
    }

    // calculate variable coordinates
    int nVars = theClass.getNClassVariables() + theClass.getNInstanceVariables();
    this.varCoords = new Point3f[nVars];
    for (int i=0; i<nVars; i++) {
        nDrawn = (float) i; // number "drawn"
        theX = 0.0f - classWidthDouble.floatValue() + LEFT_BUFFER + MEMBER_WIDTH
               + (nDrawn * 2.0f * MEMBER_WIDTH) + (nDrawn * MEMBER_BUFFER);
        theY = 0.0f - classHeightDouble.floatValue() + BOTTOM_BUFFER + methArea +
               MIDDLE_BUFFER +
               (varCohScaleFactor * varCohDiff[i] * MEMBER_HEIGHT) -
               MEMBER_HEIGHT;
        theZ = CALL_COORD_Z;
        varCoords[i] = new Point3f(theX, theY, theZ);
        //System.out.println("Var X: " + theX + " Y: " + theY + " Z: " + theZ);
    }
}

/* draws the x, y and z axes (only used in developing the visualization)
 */
private BranchGroup createAxes() {
    BranchGroup axesGroup = new BranchGroup();
```

```
    Appearance axesAppearance = new Appearance();
    LineAttributes lAttributes = new LineAttributes(1.0f,
                                 LineAttributes.PATTERN_SOLID, false);
    axesAppearance.setLineAttributes(lAttributes);

    // x axis
    LineArray axesX = new LineArray(2, LineArray.COORDINATES |
                                       LineArray.COLOR_3);
    axesX.setCoordinate(0, new Point3f(-2.0f, 0.0f, 0.0f));
    axesX.setCoordinate(1, new Point3f(2.0f, 0.0f, 0.0f));
    axesX.setColor(0, JColorConstants.REF);
    axesX.setColor(1, JColorConstants.REF);
    axesGroup.addChild(new Shape3D(axesX, axesAppearance));

    // y axis
    LineArray axesY = new LineArray(2, LineArray.COORDINATES |
                                       LineArray.COLOR_3);
    axesY.setCoordinate(0, new Point3f(0.0f, -2.0f, 0.0f));
    axesY.setCoordinate(1, new Point3f(0.0f, 2.0f, 0.0f));
    axesY.setColor(0, JColorConstants.REF);
    axesY.setColor(1, JColorConstants.REF);
    axesGroup.addChild(new Shape3D(axesY, axesAppearance));

    // z axis
    LineArray axesZ = new LineArray(2, LineArray.COORDINATES |
                                       LineArray.COLOR_3);
    axesZ.setCoordinate(0, new Point3f(0.0f, 0.0f, -2.0f));
    axesZ.setCoordinate(1, new Point3f(0.0f, 0.0f, 2.0f));
    axesZ.setColor(0, JColorConstants.REF);
    axesZ.setColor(1, JColorConstants.REF);
    axesGroup.addChild(new Shape3D(axesZ, axesAppearance));

    return axesGroup;
}

private BranchGroup createCallPointers() {
    int nDrawn = 0;
    int rowIndex = -1; // index into the rows of JCallMatrix
    Vector methodCalls = theClass.getSortedMethodCalls(); // names of all methods
                                                  // in order that they are drawn
    Vector used = theCallMatrix.getAllUsed(); // names of all members in order of
                                              // index into the columns of the
                                              // JCallMatrix

    BranchGroup callsGroup = new BranchGroup();

    // calculate the coordinates to use as endpoints for the call pointers
    this.calculateCallEndpoints();

    // for each cohesion value of methods in order of decreasing cohesion
    for (int i=methodCalls.size()-1; i>0; i--) {
        Vector v = (Vector) methodCalls.elementAt(i); // v may be empty
        if (v.size() > 0) {
            // for each method drawn
            for (int j=0; j<v.size(); j++) {
                rowIndex = used.indexOf(v.elementAt(j).toString());
                // for each member
                for (int k=0; k<used.size(); k++) {
                    // check that the method isn't referring to itself and that
                    // there is at least 1 call between this method and the other
                    // method or variable
                    if (rowIndex != k && theCallMatrix.getUses(rowIndex,k) > 0) {
                        //System.out.println("Have a line to draw with use of "
                        //    + theCallMatrix.getUses(rowIndex,k));
                        Appearance callAppearance = new Appearance();
                        LineAttributes callAttributes =
                            new LineAttributes(getCallLineWidth(
                                               theCallMatrix.getUses(rowIndex,k)),
                            LineAttributes.PATTERN_SOLID, false);
                        callAppearance.setLineAttributes(callAttributes);
                        LineArray callLine =
                            new LineArray(2,  LineArray.COORDINATES |
                                              LineArray.COLOR_3);
                        Point3f fromPoint = new Point3f(methCoordsTop[nDrawn]);
                        callLine.setCoordinate(0, fromPoint);
                        //System.out.println("nDrawn: " + nDrawn);
                        //is the destination a method or variable?
                        Point3f toPoint;
                        if (k >= theCallMatrix.getMatrixHeight()) {
                            // is a variable
                            toPoint =
```

```
                                new Point3f(varCoords[varCoordsNames.indexOf(
                                    used.elementAt(k).toString())]);
                            } else {
                                // is a method
                                toPoint = new
                                    Point3f(methCoordsBottom[methCoordsNames.indexOf(
                                            used.elementAt(k).toString())]);
                            }
                            //System.out.println(toPoint);
                            callLine.setCoordinate(1, toPoint);
                            callLine.setColor(0, JColorConstants.BLACK);
                            callLine.setColor(1, JColorConstants.BLACK);
                            callsGroup.addChild(new
                                                Shape3D(callLine, callAppearance));
                        }
                    }
                    nDrawn++;
                }
            }
        }

        return callsGroup;
    }

    /* builds the class and its members scenegraph */
    public BranchGroup createClass() {
        BranchGroup classGroup = new BranchGroup();

        // move the class
        Transform3D translate = new Transform3D();
        double adjustment = -1.0d * (getZDimension(theClass.getGrade()));
        //System.out.println("Adjustment for class in z dim: " + adjustment +
        //    " zdim: " + getZDimension(theClass.getGrade()));
        Vector3d vector = new Vector3d(0.0d, 0.0d, adjustment);
        translate.setTranslation(vector);
        TransformGroup objMove = new TransformGroup(translate);
        classGroup.addChild(objMove);

        // color and transparencify the class
        ColoringAttributes cAttributes = new ColoringAttributes();
        cAttributes.setColor(getColor(theClass.getGrade())); // color based on grade
        boxAppearance = new Appearance();
        boxAppearance.setColoringAttributes(cAttributes);
        TransparencyAttributes cTransparency =
                    new TransparencyAttributes(TransparencyAttributes.NONE, 0.0f);
        boxAppearance.setTransparencyAttributes(cTransparency);
        boxAppearance.setCapability(Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);

        // build the class
        Box box = new Box(getXDimension(), getYDimension(),
                          getZDimension(theClass.getGrade()),
                          Box.GENERATE_NORMALS, boxAppearance);
        box.setCapability(Box.ENABLE_APPEARANCE_MODIFY);
        box.setCapability(Box.GENERATE_NORMALS);

        objMove.addChild(box);

        return classGroup;
    }

    /* constructs the complexity fence used to see which elements
     * are of low quality
     */
    private BranchGroup createFence() {
        BranchGroup objRoot = new BranchGroup();

        // move fence in the z dimension
        Transform3D translate = new Transform3D();
        Vector3f vector = new Vector3f(0.0f, 0.0f, -0.717f);
        translate.setTranslation(vector);
        TransformGroup fenceMove = new TransformGroup(translate);
        objRoot.addChild(fenceMove);

        // fence object
        // color the fence
        ColoringAttributes cAttributes = new ColoringAttributes();
        cAttributes.setColor(JColorConstants.REF);
        Appearance fenceAppearance = new Appearance();
        fenceAppearance.setColoringAttributes(cAttributes);
        TransparencyAttributes fTransparency =
            new TransparencyAttributes(TransparencyAttributes.SCREEN_DOOR, 0.6f);
```

```
    fenceAppearance.setTransparencyAttributes(fTransparency);

    // build the fence
    Box fence = new Box(getXDimension()+0.3f, getYDimension()+0.3f,
        0.025f, fenceAppearance);
    fenceMove.addChild(fence);

    // optimize
    objRoot.compile();

    return objRoot;
}

/* creates the portion of the scenegraph with all the methods,
 * method extenders and labels
 */
private BranchGroup createMethods() {
    BranchGroup objRoot = new BranchGroup();
    int calls;
    String methodType;
    int methodIndex;
    String methodName;
    String visibility;
    JClassMethod jcm;
    JInstanceMethod jim;
    double grade;
    int nDrawn = 0; // number of methods drawn so far
    int startCohesion = 1000000; // arbitrarily high
    int thisCohesion;
    boolean isClass, isInstance;
    boolean isFirstCohesion = true;
    Vector methodCalls = theClass.getSortedMethodCalls();

    for (int i=methodCalls.size()-1; i>0; i--) {
        Vector v = (Vector) methodCalls.elementAt(i);
        if (v.size() > 0) {
            for (int j=0; j<v.size(); j++) {
                grade = 0.0;
                isInstance = false;
                isClass = false;

                //System.out.println("Have at least 1 method to draw");
                methodName = v.elementAt(j).toString();
                // store method names for later use
                if (this.methCoordsNames == null) {
                    int nMethods = theClass.getNClassMethods() +
                                   theClass.getNInstanceMethods();
                    methCoordsNames = new Vector(nMethods);
                    methCoordsNames.addElement(methodName);
                } else {
                    methCoordsNames.addElement(methodName);
                }

                methodType = theClass.getMemberType(methodName);
                if (methodType.equals("instance method")) {
                    methodType = "instance";
                } else {
                    methodType = "class";
                }
                methodIndex = theClass.getIndexOfMethod(methodType, methodName);
                if (methodType.equals("class")) {
                    isClass = true;
                } else {
                    isInstance = true;
                }
                if (isClass) {
                    jcm = theClass.getClassMethod(methodIndex);
                    grade = jcm.getGrade();
                    visibility = jcm.getVisibility();
                    thisCohesion = jcm.getNCallee() + jcm.getNCaller();
                    if (isFirstCohesion) {
                        isFirstCohesion = false;
                        startCohesion = thisCohesion;
                    }
                } else {
                    jim = theClass.getInstanceMethod(methodIndex);
                    grade = jim.getGrade();
                    visibility = jim.getVisibility();
                    thisCohesion = jim.getNCallee() + jim.getNCaller();
                    if (isFirstCohesion) {
                        isFirstCohesion = false;
```

```
        startCohesion = thisCohesion;
    }
}

BranchGroup methodRoot = new BranchGroup();
// move the method
Transform3D translate = new Transform3D();
float adjustment = -1.0f * (getZDimension(grade));
Vector3f vector = new Vector3f(getMethX(nDrawn),
    getMethY(nDrawn, startCohesion, thisCohesion), adjustment);
translate.setTranslation(vector);
TransformGroup objMove = new TransformGroup(translate);
methodRoot.addChild(objMove);

// color and transparencify the method
ColoringAttributes cAttributes = new ColoringAttributes();
cAttributes.setColor(getColor(grade));
Appearance methodAppearance = new Appearance();
methodAppearance.setColoringAttributes(cAttributes);
TransparencyAttributes methodTransparency = new
    TransparencyAttributes(TransparencyAttributes.NONE, 0.0f);
methodAppearance.setTransparencyAttributes(methodTransparency);
methodAppearance.setCapability(
                Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);

// build the method
Box methodBox = new Box(MEMBER_WIDTH, MEMBER_HEIGHT,
                        getZDimension(grade),
                        methodAppearance);
methodBox.setCapability(Box.ENABLE_APPEARANCE_MODIFY);
objMove.addChild(methodBox);

// move the id extenders out
Transform3D extend = new Transform3D();
float ext = getZDimension(grade) + (0.5f * EXT_LENGTH);
Vector3f vectorExt = new Vector3f(0.0f, 0.0f, ext);
extend.setTranslation(vectorExt);
TransformGroup extendOut = new TransformGroup(extend);
objMove.addChild(extendOut);

// color and transparencify the extenders
Appearance extAppearance = new Appearance();
if (visibility.equals("public")) {
    extAppearance.setColoringAttributes(pubAttributes);
} else if (visibility.equals("default")) {
    extAppearance.setColoringAttributes(defAttributes);
} else if (visibility.equals("protected")) {
    extAppearance.setColoringAttributes(protAttributes);
} else if (visibility.equals("private")) {
    extAppearance.setColoringAttributes(privAttributes);
} else {
    System.out.println("Error in visibility type: " +
                       visibility);
}
TransparencyAttributes eTransparency = new
    TransparencyAttributes(TransparencyAttributes.NONE, 0.0f);
extAppearance.setTransparencyAttributes(eTransparency);
extAppearance.setCapability(
             Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);

// draw the extenders
Box extBox = new Box(MEMBER_WIDTH,
                     MEMBER_HEIGHT, (0.5f * EXT_LENGTH),
                     extAppearance);
extBox.setCapability(Box.ENABLE_APPEARANCE_MODIFY);
extendOut.addChild(extBox);

// label the extenders
String theLabelText = null;
if (isClass) {
    jcm = theClass.getClassMethod(methodIndex);
    theLabelText = jcm.getLabel();
} else {
    jim = theClass.getInstanceMethod(methodIndex);
    theLabelText = jim.getLabel();
}

// extend label out an additional amount and scale down
Transform3D labelExt = new Transform3D();
float moreExt = getZDimension(grade) + EXT_LENGTH;
Vector3f vectorLabel = new Vector3f(0.0f, 0.0f, moreExt);
```

```
                labelExt.setTranslation(vectorLabel);

                Transform3D labelScale = new Transform3D();
                Vector3d vectorScale = new Vector3d(0.05d, 0.05d, 0.05d);
                labelScale.setScale(vectorScale);

                labelExt.mul(labelScale);
                TransformGroup labelMove = new TransformGroup(labelExt);
                objMove.addChild(labelMove);

                // label appearance
                ColoringAttributes labelColor = new ColoringAttributes();
                labelColor.setColor(JColorConstants.BLACK);
                Appearance labelAppearance = new Appearance();
                labelAppearance.setColoringAttributes(labelColor);
                labelAppearance.setTransparencyAttributes(opaque);
                labelAppearance.setCapability(
                    Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);
                Font f = new Font("SansSerif", Font.BOLD, 1);
                Font3D font3D = new Font3D(f, new FontExtrusion());
                Text3D labelText = new Text3D(font3D, new String(theLabelText));
                labelText.setAlignment(Text3D.ALIGN_CENTER);
                Shape3D labelShape = new Shape3D();
                labelShape.setGeometry(labelText);
                labelShape.setAppearance(labelAppearance);

                labelMove.addChild(labelShape);

                objRoot.addChild(methodRoot);
                nDrawn++;
            }
        }
    }

    return objRoot;
}

/* constructs the content scenegraph */
public BranchGroup createSceneGraph() {
    BranchGroup objRoot = new BranchGroup();

    // set Frame background color
    Background background = new Background(0.9f, 1.0f, 1.0f);
    background.setApplicationBounds(new BoundingSphere());
    objRoot.addChild(background);

    // set initial view
    Transform3D translate = new Transform3D();
    Vector3f vector = new Vector3f(0.0f, 0.0f, 0.0f);
    translate.setTranslation(vector);

    // set up intial view
    TransformGroup initialView = new TransformGroup(translate);
    initialView.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    initialView.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    objRoot.addChild(initialView);

    // set up ability for user to rotate class with mouse
    MouseRotate sceneRotator = new MouseRotate();
    sceneRotator.setTransformGroup(initialView);
    sceneRotator.setSchedulingBounds(new BoundingSphere());
    objRoot.addChild(sceneRotator);

    // set up ability for user to move class with mouse
    MouseTranslate sceneTranslator = new MouseTranslate();
    sceneTranslator.setTransformGroup(initialView);
    sceneTranslator.setSchedulingBounds(new BoundingSphere());
    objRoot.addChild(sceneTranslator);

    // set up ability for user to zoom class with mouse
    MouseZoom sceneZoom = new MouseZoom();
    sceneZoom.setTransformGroup(initialView);
    sceneZoom.setSchedulingBounds(new BoundingSphere());
    objRoot.addChild(sceneZoom);

    // move the center of the class to the origin
    Transform3D center = new Transform3D();
    Vector3f vector2 = new Vector3f(0.0f, 0.0f,
                            (1.0f * getZDimension(theClass.getGrade())));
    center.setTranslation(vector2);
    TransformGroup centerClass = new TransformGroup(center);
```

```
        initialView.addChild(centerClass);

        // add class
        BranchGroup classGroup = createClass();
        centerClass.addChild(classGroup);

        // add methods
        BranchGroup methodsGroup = createMethods();
        centerClass.addChild(methodsGroup);

        // add variables
        BranchGroup variablesGroup = createVariables();
        centerClass.addChild(variablesGroup);

        // add axes
        //BranchGroup axesGroup = createAxes();
        //initialView.addChild(axesGroup);
        //centerClass.addChild(axesGroup);

        // add complexity fence
        BranchGroup fenceGroup = createFence();
        centerClass.addChild(fenceGroup);

        // add call pointers
        BranchGroup callsGroup = createCallPointers();
        centerClass.addChild(callsGroup);

        // make all components "hear" changes in viewing scope
        JListener listener = new JListener(initialView);
        vScope.addItemListener(listener);
        listener.setSchedulingBounds(new
                        BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0));
        objRoot.addChild(listener);

        // optimize
        objRoot.compile();

        return objRoot;
    }

    /* constructs the scenegraph portion containing the variables
     * and the variables' ids
     */
    private BranchGroup createVariables() {
        BranchGroup objRoot = new BranchGroup();
        int calls;
        String varType;
        int varIndex;
        String varName;
        String visibility;
        JClassVariable jcv;
        JInstanceVariable jiv;
        double grade;
        int nDrawn = 0;
        int startCohesion = 1000000; // arbitrarily high
        int thisCohesion;
        boolean isClass, isInstance;
        boolean isFirstCohesion = true;
        Vector varCalls = theClass.getSortedVariableCalls();

        for (int i=varCalls.size()-1; i>0; i--) {
            Vector v = (Vector) varCalls.elementAt(i);
            if (v.size() > 0) {
                for (int j=0; j<v.size(); j++) {
                    grade = 0.0;
                    isInstance = false;
                    isClass = false;

                    //System.out.println("Have at least 1 variable to draw");
                    varName = v.elementAt(j).toString();
                    // store var names for later use
                    if (this.varCoordsNames == null) {
                        int nVars = theClass.getNClassVariables() +
                                    theClass.getNInstanceVariables();
                        varCoordsNames = new Vector(nVars);
                        varCoordsNames.addElement(varName);
                    } else {
                        varCoordsNames.addElement(varName);
                    }

                    varType = theClass.getMemberType(varName);
```

```
if (varType.equals("instance variable")) {
    varType = "instance";
} else {
    varType = "class";
}
varIndex = theClass.getIndexOfVariable(varType, varName);
if (varType.equals("class")) {
    isClass = true;
} else {
    isInstance = true;
}
if (isClass) {
    jcv = theClass.getClassVariable(varIndex);
    grade = jcv.getGrade();
    visibility = jcv.getVisibility();
    thisCohesion = jcv.getNCallee();
    if (isFirstCohesion) {
        isFirstCohesion = false;
        startCohesion = thisCohesion;
    }
} else {
    jiv = theClass.getInstanceVariable(varIndex);
    grade = jiv.getGrade();
    visibility = jiv.getVisibility();
    thisCohesion = jiv.getNCallee();
    if (isFirstCohesion) {
        isFirstCohesion = false;
        startCohesion = thisCohesion;
    }
}

BranchGroup varRoot = new BranchGroup();

// move the var
Transform3D translate = new Transform3D();
float adjustment = -1.0f * (getZDimension(grade));
Vector3f vector = new Vector3f(getVarX(nDrawn),
    getVarY(nDrawn, startCohesion, thisCohesion), adjustment);
translate.setTranslation(vector);

// rotate the var
Transform3D rotate = new Transform3D();
rotate.rotX(Math.PI/2.0d); // 90 degrees
translate.mul(rotate);

TransformGroup objMove = new TransformGroup(translate);
varRoot.addChild(objMove);

// color and transparencify the var
ColoringAttributes cAttributes = new ColoringAttributes();
cAttributes.setColor(getColor(grade));
Appearance varAppearance = new Appearance();
varAppearance.setColoringAttributes(cAttributes);
TransparencyAttributes vTransparency = new
    TransparencyAttributes(TransparencyAttributes.NONE, 0.0f);
varAppearance.setTransparencyAttributes(vTransparency);
varAppearance.setCapability(
    Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);

// build the var
Cylinder varCylinder = new Cylinder(MEMBER_WIDTH,
                       (2.0f * getZDimension(grade)),
                       Cylinder.GENERATE_NORMALS, varAppearance);
varCylinder.setCapability(Cylinder.ENABLE_APPEARANCE_MODIFY);
objMove.addChild(varCylinder);

// move the id extenders out
Transform3D extend = new Transform3D();
float ext = getZDimension(grade) + (0.5f * EXT_LENGTH);
Vector3f vectorExt = new Vector3f(0.0f, ext, 0.0f);
extend.setTranslation(vectorExt);
TransformGroup extendOut = new TransformGroup(extend);
objMove.addChild(extendOut);

// color and transparencify the extenders
Appearance extAppearance = new Appearance();
if (visibility.equals("public")) {
    extAppearance.setColoringAttributes(pubAttributes);
} else if (visibility.equals("default")) {
    extAppearance.setColoringAttributes(defAttributes);
} else if (visibility.equals("protected")) {
```

```
                    extAppearance.setColoringAttributes(protAttributes);
                } else if (visibility.equals("private")) {
                    extAppearance.setColoringAttributes(privAttributes);
                } else {
                    System.out.println("Error in visibility type: " +
                                       visibility);
                }
                TransparencyAttributes eTransparency = new
                    TransparencyAttributes(TransparencyAttributes.NONE, 0.0f);
                extAppearance.setTransparencyAttributes(eTransparency);
                extAppearance.setCapability(
                    Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);

                // draw the extenders
                Cylinder extCylinder = new Cylinder(MEMBER_WIDTH, EXT_LENGTH,
                                       Cylinder.GENERATE_NORMALS, extAppearance);
                extCylinder.setCapability(Cylinder.ENABLE_APPEARANCE_MODIFY);
                extendOut.addChild(extCylinder);

                // label the extenders
                String theLabelText = null;
                if (isClass) {
                    jcv = theClass.getClassVariable(varIndex);
                    theLabelText = jcv.getLabel();
                } else {
                    jiv = theClass.getInstanceVariable(varIndex);
                    theLabelText = jiv.getLabel();
                }

                // extend label out an additional amount and scale down
                // and rotate back 90 degrees around x axis
                Transform3D labelExt = new Transform3D();
                float moreExt = getZDimension(grade) + EXT_LENGTH;
                Vector3f vectorLabel = new Vector3f(0.0f, moreExt, 0.0f);
                labelExt.setTranslation(vectorLabel);

                Transform3D rotateLabel = new Transform3D();
                rotateLabel.rotX(Math.PI/-2.0d);
                labelExt.mul(rotateLabel);

                Transform3D labelScale = new Transform3D();
                Vector3d vectorScale = new Vector3d(0.05d, 0.05d, 0.05d);
                labelScale.setScale(vectorScale);
                labelExt.mul(labelScale);

                TransformGroup labelMove = new TransformGroup(labelExt);
                objMove.addChild(labelMove);

                // label appearance
                ColoringAttributes labelColor = new ColoringAttributes();
                labelColor.setColor(JColorConstants.BLACK);
                Appearance labelAppearance = new Appearance();
                labelAppearance.setColoringAttributes(labelColor);
                labelAppearance.setTransparencyAttributes(opaque);
                labelAppearance.setCapability(
                    Appearance.ALLOW_TRANSPARENCY_ATTRIBUTES_WRITE);
                Font f = new Font("SansSerif", Font.BOLD, 1);
                Font3D font3D = new Font3D(f, new FontExtrusion());
                Text3D labelText = new Text3D(font3D, new String(theLabelText));
                labelText.setCapability(Text3D.ALLOW_STRING_WRITE);
                labelText.setAlignment(Text3D.ALIGN_CENTER);
                Shape3D labelShape = new Shape3D();
                labelShape.setGeometry(labelText);
                labelShape.setAppearance(labelAppearance);
                labelShape.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);

                labelMove.addChild(labelShape);

                nDrawn++;
                objRoot.addChild(varRoot);
            }
        }
    }
    return objRoot;
}

public void destroy() {
    universe.removeAllLocales();
}

/* the equivalent of a main */
```

```
public void drawModel() {
    System.out.println("\tBeginning to draw " + this.theClass.getClassName() +
                       "...");
    Frame frame = new MainFrame(new JModeler(this.theClass,
                                    this.theCallMatrix), 900, 700);
}

/* determines what the call line width should be from the range of calls made
 */
private float getCallLineWidth(int nCalls) {
    float width = 1.0f;

    this.setNVariableCalls();

    int maxVarCall = this.maxVariableCall;
    int minVarCall = this.minVariableCall;
    float varRange = (float) maxVarCall - minVarCall;
    float widthRange = MAX_LINE_WIDTH - MIN_LINE_WIDTH;

    if (maxVarCall == minVarCall) {
        width = 0.5f * widthRange;
    } else {
        width = (((nCalls - minVarCall) / varRange) * widthRange) +
                MIN_LINE_WIDTH;
    }
    //System.out.println("ncalls: " + nCalls + " lineWidth: " + width);
    //System.out.println("maxCalls: " + maxVarCall + " minCalls: " + minVarCall);

    return width;
}

/* returns a color based on the grade of the element */
private Color3f getColor(double grade) {
    Color3f color = null;

    if (grade == 100.0d)       { color = JColorConstants.G100;
    } else if (grade == 95.0d) { color = JColorConstants.G95;
    } else if (grade == 90.0d) { color = JColorConstants.G90;
    } else if (grade == 85.0d) { color = JColorConstants.G85;
    } else if (grade == 80.0d) { color = JColorConstants.G80;
    } else if (grade == 75.0d) { color = JColorConstants.G75;
    } else if (grade == 70.0d) { color = JColorConstants.G70;
    } else if (grade == 65.0d) { color = JColorConstants.G65;
    } else if (grade == 60.0d) { color = JColorConstants.G60;
    } else if (grade == 55.0d) { color = JColorConstants.G55;
    } else if (grade == 50.0d) { color = JColorConstants.G50;
    } else if (grade == 45.0d) { color = JColorConstants.G45;
    } else if (grade == 40.0d) { color = JColorConstants.G40;
    } else if (grade == 35.0d) { color = JColorConstants.G35;
    } else if (grade == 30.0d) { color = JColorConstants.G30;
    } else if (grade == 25.0d) { color = JColorConstants.G25;
    } else if (grade == 20.0d) { color = JColorConstants.G20;
    } else if (grade == 15.0d) { color = JColorConstants.G15;
    } else if (grade == 10.0d) { color = JColorConstants.G10;
    } else if (grade == 5.0d)  { color = JColorConstants.G5;
    } else if (grade == 0.0d)  { color = JColorConstants.G0;
    }

    return color;
}

private float getMethX(int nDrawn) {
    Double dimension;
    Integer nDrawnInt = new Integer(nDrawn);
    double nDrawnDouble = nDrawnInt.doubleValue();

    dimension = new Double((-1.0f * classWidth) + LEFT_BUFFER + MEMBER_WIDTH +
                (nDrawnDouble * 2.0f * MEMBER_WIDTH) +
                (nDrawnDouble * MEMBER_BUFFER));
    //System.out.println("Method x dimension: " + dimension);
    return dimension.floatValue();
}

private float getMethY(int nDrawn, int startCohesion, int thisCohesion) {
    Double dimension;
    Integer nDrawnInt = new Integer(nDrawn); // number of methods already drawn
    double nDrawnDouble = nDrawnInt.doubleValue(); // number of meths drawn
    int cohesionDifference; // diff in cohesion value between last method drawn
                            // and the method currently being drawn
    cohesionDifference = startCohesion - thisCohesion;
```

```
    // add to (or start) the storage of the method cohesion differences so that
    // they can be used later to draw the method calls
    if (methCohDiff == null) {
        int nMethods = theClass.getNClassMethods() +
                       theClass.getNInstanceMethods();
        methCohDiff = new int [nMethods];
        methCohDiff[0] = 0; // first method has no previous meth to compare it to
    } else {
        methCohDiff[nDrawn] = cohesionDifference;
    }

    // think about the placement this way: move the method all the way to the top
    // of the class and then start moving it back down til its in the right spot
    dimension = new Double(classHeight - TOP_BUFFER - varArea -
                MIDDLE_BUFFER  -
                (methCohScaleFactor * cohesionDifference * MEMBER_HEIGHT));

   // System.out.println("Method y dimension: " + dimension + " startCohesion: "
   //      + startCohesion + " thisCohesion: " + thisCohesion +
   //      " cohesion diff: " + cohesionDifference);

    return dimension.floatValue();
}

private float getVarX(int nDrawn) {
    Double dimension;
    Integer nDrawnInt = new Integer(nDrawn);
    double nDrawnDouble = nDrawnInt.doubleValue();

    dimension = new Double((-1.0f * classWidth) + LEFT_BUFFER + MEMBER_WIDTH +
                (nDrawnDouble * 2.0f * MEMBER_WIDTH) +
                (nDrawnDouble * MEMBER_BUFFER));
    //System.out.println("Var x dimension: " + dimension);
    return dimension.floatValue();
}

private float getVarY(int nDrawn, int startCohesion, int thisCohesion) {
    Double dimension;
    Integer nDrawnInt = new Integer(nDrawn); // number of vars already drawn
    double nDrawnDouble = nDrawnInt.doubleValue(); // number of vars drawn
    int cohesionDifference; // diff in cohesion value between last var drawn
                            // and the var currently being drawn
    cohesionDifference = startCohesion - thisCohesion;

    // add to (or start) the storage of the var cohesion differences so that they
    // can be used later to draw the var uses (aka method calls)
    if (varCohDiff == null) {
        int nVars = theClass.getNClassVariables() +
                    theClass.getNInstanceVariables();
        varCohDiff = new int [nVars];
        varCohDiff[0] = 0; // first var has no previous var to compare it to
    } else {
        varCohDiff[nDrawn] = cohesionDifference;
    }

    // think about the placement this way: move the var all the way to the bottom
    // of the class and then start moving it back up til its in the right spot
    dimension = new Double((-1.0f * classHeight) + BOTTOM_BUFFER + methArea +
                    MIDDLE_BUFFER +
                    (varCohScaleFactor * cohesionDifference * MEMBER_HEIGHT));

    //System.out.println("Var y dimension: " + dimension + " startCohesion: " +
                       // startCohesion + " thisCohesion: " + thisCohesion +
                       // " cohesion diff: " + cohesionDifference);

    return dimension.floatValue();
}

/* calculates the width of the class which is based mainly on the
 * max {(num Vars * var width), (num meths + meth width)}
 */
private float getXDimension() {
    float dimension;
    float centerArea;
    int nMethods = theClass.getNClassMethods() + theClass.getNInstanceMethods();
    int nVariables = theClass.getNClassVariables() +
                     theClass.getNInstanceVariables();
    int maxMembers;
    if (nMethods > nVariables) {
        maxMembers = nMethods;
    } else {
```

```
        maxMembers = nVariables;
    }
    // centerArea: max(method widths, var widths) + spacers
    centerArea = (maxMembers * MEMBER_WIDTH) + ((maxMembers-1) * MEMBER_BUFFER);

    dimension = LEFT_BUFFER + centerArea + RIGHT_BUFFER;

    this.classWidth = dimension;

    return dimension;
}

/* calculates the height of the class which is based on the
 * range of the class to variables + the range of the calls
 * from methods
 */
private float getYDimension() {
    float dimension;

    if (theClass.getVariableCohesionRange()+1 > MAX_COHESION_DIFFERENCE) {
        this.varCohScaleFactor = 1.0f;
    } else {
        this.varCohScaleFactor = 1.0f;
    }
    if (theClass.getMethodCohesionRange()+1 > MAX_COHESION_DIFFERENCE) {
        this.methCohScaleFactor = 1.0f;
    } else {
        this.methCohScaleFactor = 1.0f;
    }

    float varArea = (varCohScaleFactor *
                    (theClass.getVariableCohesionRange()+1.0f)) * MEMBER_HEIGHT;
    this.varArea = varArea;

    float methArea = (methCohScaleFactor *
                     (theClass.getMethodCohesionRange()+1.0f)) * MEMBER_HEIGHT;
    this.methArea = methArea;

    dimension = 0.5f * (TOP_BUFFER + varArea + MIDDLE_BUFFER + methArea +
                        BOTTOM_BUFFER);
    //System.out.println("in getYdim(): TOPB: " + TOP_BUFFER + " VARS: " +
    //    varArea + " MIDB: " + MIDDLE_BUFFER + " METHS: " + methArea +
    //    " BOTB: " + BOTTOM_BUFFER + " classHeight: " + dimension);
    this.classHeight = dimension;

    return dimension;
}

/* calculates the z dimension of the class based on the grade of the class */
private float getZDimension(double grade) {
    Double g= new Double(grade);
    float floatGrade = g.floatValue();
    float maxGrade = 100.0f;
    return 0.3f + (.015f * ((maxGrade - floatGrade) / 5.0f));
}

/* sets up everything needed to start the GUI
 * called automatically when this class is initialized
*/
public void init() {
    this.classTransparency = 0.0f; // start out in out-public scope

    defAttributes = new ColoringAttributes(); // for label extenders
    defAttributes.setCapability(ColoringAttributes.ALLOW_COLOR_WRITE);
    defAttributes.setColor(JColorConstants.LTGRAY);

    privAttributes = new ColoringAttributes(); // for label extenders
    privAttributes.setCapability(ColoringAttributes.ALLOW_COLOR_WRITE);
    privAttributes.setColor(JColorConstants.BLACK);

    protAttributes = new ColoringAttributes(); // for label extenders
    protAttributes.setCapability(ColoringAttributes.ALLOW_COLOR_WRITE);
    protAttributes.setColor(JColorConstants.BLACK);

    pubAttributes = new ColoringAttributes(); // for label extenders
    pubAttributes.setCapability(ColoringAttributes.ALLOW_COLOR_WRITE);
    pubAttributes.setColor(JColorConstants.LTGRAY);

    // add choices to view scope pull-down menu (top of GUI)
    vScope.add(OUT_PUBLIC);
    vScope.add(OUT_PACKAGE);
```

```
vScope.add(OUT_SUBCLASS);
vScope.add(IN_CLASS);

// set up instruction are in lower left of GUI
tArea = new TextArea(textMessage, 5, 40,
                     TextArea.SCROLLBARS_NONE); // bottom area
tArea.setEditable(false);

setLayout(new BorderLayout(10,10));

// set up the 3D area
GraphicsConfiguration config =
   SimpleUniverse.getPreferredConfiguration();
Canvas3D canvas3D = new Canvas3D(config);
BranchGroup scene = createSceneGraph();
universe = new SimpleUniverse(canvas3D);
universe.getViewingPlatform().setNominalViewingTransform();
universe.addBranchGraph(scene);

// set up general things about the GUI
backPanel = new Panel();
backPanel.setVisible(true);
backPanel.setLayout(new BorderLayout());

// set up the view scope message at top of GUI
scopeLabel = new Label("Current view is from: ", Label.LEFT);

// set up the statistical report at right-center of GUI
reportArea = new TextArea(this.getReport(), 35, 50,
                          TextArea.SCROLLBARS_VERTICAL_ONLY);

// set up the top of the GUI
northPanel = new Panel();
northPanel.setLayout(new FlowLayout());
northPanel.add(scopeLabel);
northPanel.add(vScope);
northPanel.setBackground(JColorConstants.GRAY);
backPanel.add(northPanel, BorderLayout.NORTH);

// set up the center of GUI (3D and report)
centralPanel = new Panel(new GridLayout(1,2,10,10));
centralPanel.add(canvas3D);
centralPanel.add(reportArea);
centralPanel.setBackground(JColorConstants.GRAY);
backPanel.add(centralPanel, BorderLayout.CENTER);

// set up the bottom of GUI
southPanel = new Panel(new GridLayout(1,2,5,880));
southPanel.add(tArea);

// build color panel
Panel colorPanel = new Panel();
colorPanel.setLayout(new GridLayout(4,11));
// 1st row of color panel
Label keyLabel = new Label("Key:");
colorPanel.add(keyLabel);
Panel color1 = new Panel();
color1.setBackground(JColorConstants.dkGreen);
colorPanel.add(color1);
Panel color2 = new Panel();
color2.setBackground(JColorConstants.dkGreen_green);
colorPanel.add(color2);
Panel color3 = new Panel();
color3.setBackground(JColorConstants.green);
colorPanel.add(color3);
Panel color4 = new Panel();
color4.setBackground(JColorConstants.green_greenYellow);
colorPanel.add(color4);
Panel color5 = new Panel();
color5.setBackground(JColorConstants.green_yellow);
colorPanel.add(color5);
Panel color6 = new Panel();
color6.setBackground(JColorConstants.greenYellow_yellow);
colorPanel.add(color6);
Panel color7 = new Panel();
color7.setBackground(JColorConstants.yellow);
colorPanel.add(color7);
Panel color8 = new Panel();
color8.setBackground(JColorConstants.yellow_yellowOrange);
colorPanel.add(color8);
Panel color9 = new Panel();
```

```
color9.setBackground(JColorConstants.yellow_orange);
colorPanel.add(color9);
Panel color10 = new Panel();
color10.setBackground(JColorConstants.yellowOrange_orange);
colorPanel.add(color10);
// 2nd row of color panel
Label scoreLabel = new Label("Score:");
colorPanel.add(scoreLabel);
Label lColor1 = new Label("100", Label.CENTER);
colorPanel.add(lColor1);
Label lColor2 = new Label("95", Label.CENTER);
colorPanel.add(lColor2);
Label lColor3 = new Label("90", Label.CENTER);
colorPanel.add(lColor3);
Label lColor4 = new Label("85", Label.CENTER);
colorPanel.add(lColor4);
Label lColor5 = new Label("80", Label.CENTER);
colorPanel.add(lColor5);
Label lColor6 = new Label("75", Label.CENTER);
colorPanel.add(lColor6);
Label lColor7 = new Label("70", Label.CENTER);
colorPanel.add(lColor7);
Label lColor8 = new Label("65", Label.CENTER);
colorPanel.add(lColor8);
Label lColor9 = new Label("60", Label.CENTER);
colorPanel.add(lColor9);
Label lColor10 = new Label("55", Label.CENTER);
colorPanel.add(lColor10);
// 3rd row of color panel
Panel color11 = new Panel();
color11.setBackground(JColorConstants.orange);
colorPanel.add(color11);
Panel color12 = new Panel();
color12.setBackground(JColorConstants.orange_orangeRed);
colorPanel.add(color12);
Panel color13 = new Panel();
color13.setBackground(JColorConstants.orange_red);
colorPanel.add(color13);
Panel color14 = new Panel();
color14.setBackground(JColorConstants.orangeRed_red);
colorPanel.add(color14);
Panel color15 = new Panel();
color15.setBackground(JColorConstants.red);
colorPanel.add(color15);
Panel color16 = new Panel();
color16.setBackground(JColorConstants.red_dkishred);
colorPanel.add(color16);
Panel color17 = new Panel();
color17.setBackground(JColorConstants.dkishred);
colorPanel.add(color17);
Panel color18 = new Panel();
color18.setBackground(JColorConstants.dkishred_dkred);
colorPanel.add(color18);
Panel color19 = new Panel();
color19.setBackground(JColorConstants.dkred);
colorPanel.add(color19);
Panel color20 = new Panel();
color20.setBackground(JColorConstants.dkred_verydkred);
colorPanel.add(color20);
Panel color21 = new Panel();
color21.setBackground(JColorConstants.verydkred);
colorPanel.add(color21);
// 4th row of color panel
Label lColor11 = new Label("50", Label.CENTER);
colorPanel.add(lColor11);
Label lColor12 = new Label("45", Label.CENTER);
colorPanel.add(lColor12);
Label lColor13 = new Label("40", Label.CENTER);
colorPanel.add(lColor13);
Label lColor14 = new Label("35", Label.CENTER);
colorPanel.add(lColor14);
Label lColor15 = new Label("30", Label.CENTER);
colorPanel.add(lColor15);
Label lColor16 = new Label("25", Label.CENTER);
colorPanel.add(lColor16);
Label lColor17 = new Label("20", Label.CENTER);
colorPanel.add(lColor17);
Label lColor18 = new Label("15", Label.CENTER);
colorPanel.add(lColor18);
Label lColor19 = new Label("10", Label.CENTER);
colorPanel.add(lColor19);
```

```
        Label lColor20 = new Label("5", Label.CENTER);
        colorPanel.add(lColor20);
        Label lColor21 = new Label("0", Label.CENTER);
        colorPanel.add(lColor21);

        // finish setting up bottom of GUI
        southPanel.add(colorPanel);
        southPanel.setBackground(JColorConstants.GRAY);

        // finish setting up GUI
        backPanel.add(southPanel, BorderLayout.SOUTH);
        backPanel.setBackground(JColorConstants.GRAY);
        add(backPanel);
    }

    /* returns a statistical report to print in the center-right of the GUI */
    private String getReport() {
        StringBuffer buffer = new StringBuffer();
        Vector minGradeMembers;
        Vector criticisms;
        String type;

        // Elements needing the most attention
        minGradeMembers = theClass.getMinimumGradeMembers();
        buffer.append("Element(s) within " + theClass.getClassName() +
                      " needing the most attention:\n");
        for (int i=0; i<minGradeMembers.size(); i++) {
            type = theClass.getMemberType(minGradeMembers.elementAt(i).toString());
            buffer.append("\t" + type + " " + minGradeMembers.elementAt(i) + " (");
            buffer.append(
                theClass.getLabel(minGradeMembers.elementAt(i).toString()));
            buffer.append(") (score: " + theClass.getMinimumGrade() + ")\n");
        }

        // Class information
        buffer.append("\n-----------------------------------------------------------------
------------");
        buffer.append("\nCLASS:\n");
        buffer.append(theClass); // print a summary of class information
        criticisms = theClass.getCriticism();
        buffer.append("\n\nCriticism for " + theClass.getClassName() + ":\n");
        for (int i=0; i<criticisms.size(); i++) {
            buffer.append("\t" + criticisms.elementAt(i) + "\n");
        }
        if (criticisms.size() == 0) {
            buffer.append("\tDo not have any criticism\n");
        }

        // Class methods information
        buffer.append("\n-----------------------------------------------------------------
------------");
        buffer.append("\nCLASS METHODS:\n");
        for (int i=0; i<theClass.getNClassMethods(); i++) {
            buffer.append("\n" + theClass.getClassMethod(i));
            criticisms = theClass.getClassMethod(i).getCriticism();
            buffer.append("\n\nCriticism for " +
                          theClass.getClassMethod(i).getMethodName());
            buffer.append(" (" + theClass.getClassMethod(i).getLabel() + ") :\n");
            for (int j=0; j<criticisms.size(); j++) {
                buffer.append("\t" + criticisms.elementAt(j) + "\n");
            }
            if (criticisms.size() == 0) {
                buffer.append("\tDo not have any criticism\n");
            }
        }
        if (theClass.getNClassMethods() == 0) {
            buffer.append("\n\nNo class methods");
        }

        // Instance methods information
        buffer.append("\n-----------------------------------------------------------------
------------");
        buffer.append("\nINSTANCE METHODS:\n");
        for(int i=0; i<theClass.getNInstanceMethods(); i++) {
            buffer.append("\n" + theClass.getInstanceMethod(i));
            criticisms = theClass.getInstanceMethod(i).getCriticism();
            buffer.append("\n\nCriticism for " +
                          theClass.getInstanceMethod(i).getMethodName());
            buffer.append(" (" + theClass.getInstanceMethod(i).getLabel() + ") :\n");
            for (int j=0; j<criticisms.size(); j++) {
                buffer.append("\t" + criticisms.elementAt(j) + "\n");
```

```
            }
            if (criticisms.size() == 0) {
                buffer.append("\tDo not have any criticism\n");
            }
        }
        if (theClass.getNInstanceMethods() == 0) {
            buffer.append("\n\nNo instance methods");
        }

        // Class variables information
        buffer.append("\n-------------------------------------------------------------------
------------");
        buffer.append("\nCLASS VARIABLES:\n");
        for (int i=0; i<theClass.getNClassVariables(); i++) {
            buffer.append("\n" + theClass.getClassVariable(i));
            criticisms = theClass.getClassVariable(i).getCriticism();
            buffer.append("\n\nCriticism for " +
                          theClass.getClassVariable(i).getVariableName());
            buffer.append(" (" + theClass.getClassVariable(i).getLabel() + ") :\n");
            for (int j=0; j<criticisms.size(); j++) {
                buffer.append("\t" + criticisms.elementAt(j) + "\n");
            }
            if (criticisms.size() == 0) {
                buffer.append("\tDo not have any criticism\n");
            }
        }
        if (theClass.getNClassVariables() == 0) {
            buffer.append("\n\nNo class variables");
        }

        // Instance variables information
        buffer.append("\n-------------------------------------------------------------------
------------");
        buffer.append("\nINSTANCE VARIABLES:\n");
        for (int i=0; i<theClass.getNInstanceVariables(); i++) {
            buffer.append("\n" + theClass.getInstanceVariable(i));
            criticisms = theClass.getInstanceVariable(i).getCriticism();
            buffer.append("\n\nCriticism for " +
                          theClass.getInstanceVariable(i).getVariableName());
            buffer.append(" (" + theClass.getInstanceVariable(i).getLabel() +
                          ") :\n");
            for (int j=0; j<criticisms.size(); j++) {
                buffer.append("\t" + criticisms.elementAt(j) + "\n");
            }
            if (criticisms.size() == 0) {
                buffer.append("\tDo not have any criticism\n");
            }
        }
        if (theClass.getNInstanceVariables() == 0) {
            buffer.append("\nNo instance variables");
        }

        // decoration
        buffer.append("\n\n-------------------------------------------------------------------
--------------");

        return buffer.toString();
    }

    /* finds the min & max number of calls between a method and any other
     * method or variable
     */
    private void setNVariableCalls() {
        int calls;
        int minUse = 10000000;
        int maxUse = 0;
        int [][] uses = theCallMatrix.getAllUses();

        for (int i=0; i<theCallMatrix.getMatrixHeight(); i++) {
            for (int j=0; j<theCallMatrix.getMatrixWidth(); j++) {
                calls = uses[i][j];
                if (calls > 0 && calls < minUse) {
                    minUse = calls;
                }
                if (calls > maxUse) {
                    maxUse = calls;
                }
            }
        }

        if (minUse == 10000000) {minUse = 0;}
```

```
        this.minVariableCall = minUse;
        this.maxVariableCall = maxUse;
    }

}
```

JParseState.java

```
/**
 * JParseState
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JParseState {

    /* instance variables */
    private boolean inComment;
    private boolean inMethodDeclaration;
    private boolean inMethodDefinition;
    private boolean inVariableDeclaration;
    private boolean isConstructor;
    private String lastContent; // used to locate method documentation
    private String methodType; // "neither", "class", or "instance"
    private int nMLeftBraces; // number of left braces seen in a method
    private int nMRightBraces; // number of right braces seen in a method
    private JGrammar grammar; // can recognize Java grammar (types, modifiers, etc.)

    public JParseState() {
        this.inComment = false;
        this.inMethodDeclaration = false;
        this.inMethodDefinition = false;
        this.inVariableDeclaration = false;
        this.isConstructor = false;
        this.lastContent = "notComment";
        this.methodType = "neither";
        this.nMLeftBraces = 0;
        this.nMRightBraces = 0;
        this.grammar = new JGrammar();
    }

    /**
     * Determines if the parser is still in the middle of a multi-line
     * comment (which start with '/' and '*' and end with '*' and '/')
     *
     * @param          line    a line from the source code
     */
    public void checkInComment(String line) {
        String trimLine = line.trim();

        if ((trimLine.indexOf('*',trimLine.length()-2) == trimLine.length()-2) &&
                (trimLine.indexOf('/',trimLine.length()-1) == trimLine.length()-1)) {
            this.inComment = false; // comment closes
        } else {
            this.inComment = true; // inside of multiline comment
           }
    }

    /**
     * Determines if the parser is still in the middle of a multi-line
     * method declaration (has not seen the closing parenthesis)
     *
     * @param          line    a line from the source code
     */
    public void checkInMethodDeclaration(String line) {
        String trimLine = line.trim();

        if (trimLine.indexOf(")") == -1) {
            this.inMethodDeclaration = true;
        } else {
            this.inMethodDeclaration = false;
        }
        //System.out.println("Checked in meth decln: " + line + "\t" +
```

```
    //    this.inMethodDeclaration);
}

/**
 * Determines if the parser is still in the middle of a multi-line
 * method definition (has not seen the closing brace)
 *
 * @param          line    a line from the source code
 */
public void checkInMethodDefinition(String line) {
    String trimLine = line.trim();

    if (trimLine.indexOf("}") == -1) {
        this.inMethodDefinition = true;
    } else {
        this.inMethodDefinition = false;
    }

    //System.out.println("Checked in meth defn: " + line + "\t" +
    //    this.inMethodDefinition);
}

public void checkInVariableDeclaration(String line) {
    String trimLine = line.trim();

    if (trimLine.indexOf(";") == -1) {
        this.inVariableDeclaration = true;
    } else {
        this.inVariableDeclaration = false;
    }
}

public void countMethodBraces(String line) {
    String trimLine = line.trim();
    String token = null;
    String lastToken = null;
    StringTokenizer st = new StringTokenizer(trimLine, " \t:,(){}\"", true);

    while (st.hasMoreTokens()) {
        token = st.nextToken();

        if (token.length() > 1 && token.charAt(0) == '\\' &&
           (token.charAt(1) == '\\' || token.charAt(1) == '*')) {
            break; // otherwise it would count a brace in a comment
        }

        if (token.equals("{")) {
            if (lastToken == null || !lastToken.equals("\"")) {
                // otherwise it would include "{"
                nMLeftBraces++;
            }
        } else if (token.equals("}")) {
            if (lastToken == null || !lastToken.equals("\"")) {
                // otherwise it would include "}"
                nMRightBraces++;
            }
        }

        lastToken = token;
    }

    //System.out.println("Now have leftbraces: " + nMLeftBraces +
    //    " and rightBraces: " + nMRightBraces + " At: " + trimLine);
}

/**
 * Given a line of code, determines if this tells the class' name
 *
 * @param          line    A line of source code
 * @return         String  class name or "none"
 */
public String getClassName(String line) {
    String name = "none";
    String token = null;
    String trimLine = line.trim();
    StringTokenizer st = null;

    st = new StringTokenizer(trimLine, "\t :;()");

    while (st.hasMoreTokens()) {
        token = st.nextToken();
```

```
        if (token.toLowerCase().equals("class") && st.hasMoreTokens()) {
            name = st.nextToken();
        }
    }
    return name;
}

public boolean getInComment() {
    return inComment;
}

public boolean getInMethodDeclaration() {
    return inMethodDeclaration;
}

public boolean getInMethodDefinition() {
    return inMethodDefinition;
}

public boolean getInVariableDeclaration() {
    return inVariableDeclaration;
}

public boolean getIsConstructor() {
    return this.isConstructor;
}

public String getLastContent() {
    return this.lastContent;
}

public boolean getMethodDoc(String line) {
    return (this.lastContent.equals("comment"));
}

public String getMethodName(String line, String className) {
    String name = "none";
    boolean seenIdentifier = false;
    boolean seenType = false;
    StringTokenizer st = new StringTokenizer(line, " \t(");
    String token = st.nextToken();

    while (token != null && st.hasMoreTokens()) {
        // method name will be the first identifier after a type or the
        // second identifier if no type is seen or the class name if its
        // a constructor
        // ex: boolean method or SpecialClass method
        if (token.equals(className)) {
            name = token;
            this.isConstructor = true;
            break;
        } else if (seenType) {
            name = token;
            break;
        } else if (this.grammar.isType(token) || token.equals("void")) {
            seenType = true;
        } else if (seenIdentifier) {
            name = token;
            break;
        } else if (!token.equals("void") && !this.grammar.isModifier(token) &&
                   !this.grammar.isType(token)) {
            seenIdentifier = true;
        }

        token = st.nextToken();
    }
    return name;
}

public String getMethodType() {
    return this.methodType;
}

public String getMoreParameters(String line, JClass theClass) {
    String result = "";
    String trimLine = line.trim();
    StringTokenizer st = new StringTokenizer(trimLine, " \t");
    String token;

    while (st.hasMoreTokens()) {
        token = st.nextToken();
```

```
        if (token.indexOf(",") == -1 && token.indexOf(")") == -1 &&
                                        !token.equals("{")) {
            result = result.concat(token + "_");
        }
    }

    return result;
}

public int getNMLeftBraces() {
    return nMLeftBraces;
}

public int getNMRightBraces() {
    return nMRightBraces;
}

/**
 * returns the package name of the class or "none" the line does not
 * show the class to be in a package
 *
 * @param          line    A source code line
 * @return         String  package name or "none"
 */
public String getPackageName(String line) {
    String name = "none";
    String token = null;
     String trimLine = line.trim();
    StringTokenizer st = null;

    st = new StringTokenizer(trimLine, "\t :;()");

    while (st.hasMoreTokens()) {
        token = st.nextToken();
       if (token.toLowerCase().equals("package") && st.hasMoreTokens()) {
           name = st.nextToken();
       }
    }
    return name;
}

/**
 * For a constructor it returns an _ delimited string of parameters to
 * concatenate to the constructor name
 *
 * @param          line    A source code line
 * @return         String  package name or "none"
 */
public String getParameters(String line) {
    String result = "_";
    String trimLine = line.trim();
    StringTokenizer st = new StringTokenizer(trimLine, " (),", true);
    String token = null;
    String lastToken = "-1";
    String lastContent = "-1"; // last non-blank token

    while (st.hasMoreTokens()) {
        token = st.nextToken();

        if (token.equals(" ")) {
            // skip it
        } else if (token.equals(")")) {
            break; // no mmore parameters
       } else if ((lastToken.equals("(") || lastToken.equals(",")) &&
                    !token.equals(" ")) {
            result = result.concat(token + "_");
            lastContent = token;
        } else if (lastContent.equals("(") || lastContent.equals(",")) {
            result = result.concat(token + "_");
            lastContent = token;
        } else {
            lastContent = token;
        }

        lastToken = token;
    }

    return result;
}
```

```
public boolean getVariableDoc(String line) {
    boolean result = false;

    if (this.hasInlineComment(line)) {
        result = true;
    }

    return result;
}

/**
 * Returns the name of the variable declared on that line
 * TODO: Add the ability to detect more than one variable declaration per line
 *
 * @param         line    source code line
 * @return        String  variable declared on that line
 */
public String getVariableName(String line) {
    String name = "none";
    boolean seenIdentifier = false;
    boolean seenType = false;
    StringTokenizer st = new StringTokenizer(line, " \t(");
    String token;

    while (st.hasMoreTokens()) {
        // variable name will be the first identifier after a type or the
        // second identifier if no type is seen
        // ex: int variable or SpecialClass variable or int [] variable
        token = st.nextToken();

        if (this.grammar.isModifier(token) || token.equals("[]") ||
                                              token.equals("[][]")) {
            // skip over modifiers and array braces
        } else if (seenType && !token.equals("[]") && !token.equals("[][]")) {
            name = token;
            break;
        } else if (this.grammar.isType(token)) {
            seenType = true;
        } else if (seenIdentifier && !token.equals("[]") &&
                                     !token.equals("[][]")) {
            name = token;
            break;
        } else if (!this.grammar.isModifier(token) &&
                   !this.grammar.isType(token) && !token.equals("[]") &&
                   !token.equals("[][]")) {
            seenIdentifier = true;
            //System.out.println("Saw an identifier: " + token);
        } else {
            System.out.println("Error in getVariableName()> Line: " + line);
            System.out.println("Could not place this token: " + token);
        }

    }

    // clean up name
    if (name.indexOf(";") != -1) {
        name = name.substring(0, name.indexOf(";"));
    }

    return name;
}

public String getVisibility(String line) {
    String vis = "default";
    StringTokenizer st = new StringTokenizer(line, " \t");
    String token = st.nextToken();

    while (token != null && st.hasMoreTokens()) {
        if (token.toLowerCase().equals("public") ||
            token.toLowerCase().equals("protected") ||
            token.toLowerCase().equals("private")) {
            vis = token.toLowerCase();
            break; // take the first occurrence of an access modifier
        }

        token = st.nextToken();
    }

    return vis;
}
```

```
/**
 * Determines if any part of the line has a comment
 *
 * @param          line       source code line
 * @return         boolean    true if it has a comment on the line
 */
public boolean hasInlineComment(String line) {
    boolean result = false; // assume not comment
    String trimLine = line.trim(); // remove white space before and after line
    StringTokenizer st = new StringTokenizer(trimLine, "\t");
    String token = null;

    while (st.hasMoreTokens() && !result) {
        token = st.nextToken();

        if (token.length() > 1 && token.indexOf("//") != -1 &&
               (token.indexOf("\"") == -1 ||
                token.indexOf("\"") > token.indexOf("//"))) {
            result = true;
        } else if (token.length() > 1 && token.indexOf("/*") != -1 &&
                     (token.indexOf("\"") == -1 ||
                      token.indexOf("\"") > token.indexOf("/*"))) {
            result = true;
            // look to see if comment is closed
            this.checkInComment(trimLine);
        }
    }

    return result;
}

/**
 * Determines if the line is blank
 *
 * @param          line       A source code line
 * @return         boolean    true if its blank, else false
 */
public boolean isBlank(String line) {
    boolean result = false;
    StringTokenizer st = new StringTokenizer(line, " \t");

    if (st.countTokens() == 0) {
        result = true;
    }

    return result;
}

public boolean isFinal(String line) {
    boolean result = false;
    StringTokenizer st = new StringTokenizer(line, " \t");
    String token = st.nextToken();

    while (token != null && st.hasMoreTokens()) {
        if (token.toLowerCase().equals("final")) {
            result = true;
        }

        token = st.nextToken();
    }

    return result;
}

/**
 * Determines if the entire line is a comment
 *
 * @param          line       source code line
 * @return         boolean    true if entire line is a comment
 */
public boolean isFullLineComment(String line) {
    boolean result = false; // assume not comment
    String trimLine = line.trim(); // remove white space before and after line

    if ((trimLine.length() > 1) && (trimLine.indexOf('/') == 0) &&
                                   (trimLine.indexOf('/',1) == 1)) {
        // line starts with '//'
        result = true;
   } else if ((trimLine.length() > 1) && (trimLine.indexOf('/') == 0) &&
                                         (trimLine.indexOf('*',1) == 1)) {
        // line starts with '/*'
```

```
        result = true;
        // look to see if the comment is closed
        this.checkInComment(trimLine);
    }
    return result;
}

/**
 * Determines if this is a method declaration
 *
 * @param         line    A source code line
 * @return        String  package name or "none"
 */
public boolean isMethodDeclaration (String line) {
    boolean result = false;
    boolean sawClassName = false;
    boolean sawReturn = false;
    boolean sawVisibility = false;
    String token = null;
    String trimLine = line.trim();

    StringTokenizer st = new StringTokenizer(trimLine, " \t,", true);
    // get first token
    token = st.nextToken();

    // go through line until reaching an end-of-line comment and look for a
    // method's opening parentheses
    while (!token.equals("//") && !token.equals("/*") &&
                                    !token.equals("{")) {
        if (token.indexOf("(") != -1 && (token.indexOf("\"") == -1 ||
                                token.indexOf("\"") > token.indexOf("("))) {
            // Has a method's opening parenthesis and either does not have a
            // comment or the comment starts after the opening parenthesis
            result = true;
            // look to see if method declaration finishes
            this.checkInMethodDeclaration(trimLine);
            // look to see if method definition finishes
            if (!this.inMethodDeclaration) {
                this.checkInMethodDefinition(trimLine);
            }
        } else {
            // skip token
        }
        if (st.hasMoreTokens())
            token = st.nextToken();
        else break;
    }
    //System.out.println(trimLine);
    //System.out.println("inMethodDeclaration: " + inMethodDeclaration);
    //System.out.println("inMethodDefinition: " + inMethodDefinition);

    return result;
}

public boolean isMethodEnd() {
    boolean result = false;

    if (nMLeftBraces == nMRightBraces &&
           (nMLeftBraces != 0 && nMRightBraces != 0)) {
        result = true;
        // reset all variables pertaining to methods
        this.nMLeftBraces = 0;
        this.nMRightBraces = 0;
        this.lastContent = "notComment";
        this.inMethodDefinition = false;
        this.isConstructor = false;
    }
    return result;
}

/**
 * Determines if the program has been completely parsed
 *
 * @param         line      A source code line
 * @return        boolean   true if its static, else false
 */
public boolean isProgramEnd(String line) {
    boolean result = false;
    String trimLine = line.trim();
    StringTokenizer st = new StringTokenizer(trimLine, "} /", true);
    String token = st.nextToken();
```

```
        if (token != null && token.equals("}")) {
            result = true;
            //System.out.println("Saw last program brace.");
        }

        return result;
    }

    /**
     * Determines if the variable/method is static
     *
     * @param         line      A source code line
     * @return        boolean   true if its static, else false
     */
    public boolean isStatic (String line) {
        boolean result = false;
        String trimLine = line.trim();
        StringTokenizer st = new StringTokenizer(trimLine, " \t");
        String token = st.nextToken();

        while (token != null && st.hasMoreTokens()) {
            if (token.toLowerCase().equals("static")) {
                result = true;
            }
            token = st.nextToken();
        }

        return result;
    }

    /**
     * Determines if this is a class/instance variable declaration
     *
     * @param         line    A source code line
     * @return        String  package name or "none"
     */
    public boolean isVariableDeclaration (String line) {
        boolean result = true; // must be because it was already checked if it
                               // was a method declaration, comment or blank line
        String trimLine = line.trim();

        // see if it continues
        this.checkInVariableDeclaration(trimLine);

        return result;
    }

    public void setInMethodDeclaration(boolean isIn) {
        this.inMethodDeclaration = isIn;
    }

    public void setLastContent(String theContent) {
        this.lastContent = theContent;
    }

    public void setMethodType(String type) {
        this.methodType = type.toLowerCase();
    }

}
```

JParser.java

```
/**
 * JParser
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.io.*;

public class JParser {

    /* Instance Variables */
    private int lineNumber; // current parse line in source code
```

```
private int programScope; // 0: outside the class
                          // 1: top-level of class
                         // 2: in a method
private JClass theClass; // class to visualize
private JParseState parseState; // state of the parser (in a method, etc.)

/* Public Methods */

/**
 * Class constructor
 */
public JParser() {
    this.programScope = 0; // start outside the class
    this.lineNumber = 0; // start before line 1
    this.theClass = null;
    this.parseState = new JParseState();
}

/**
 * Parses a Java source code file and builds a JClass object
 *
 * @param         inFile  file to parse
 * @return        JClass  data structure holding file's properties
 */
public JClass parseFile(String inFile) {
    System.out.println("\tStarting to parse " + inFile + "... ");

    theClass = new JClass();
    BufferedReader reader = null;
    String currentLine = null; // current line in file
    String cName = "none"; // no default class name
    String mName = "none"; // method name
    String moreParameters; // used for multi-line Constructor declarations
    String mVisibility = "default"; // method visibility
    String pName = "none"; // assume no package
    String vName = "none"; // variable name
    String vVisibility = "default"; // variable visibility
    boolean foundClassDoc = false; // found the class documentation already
    boolean foundMDoc = false; // found the method documentation
    boolean foundVDoc = false; // found the variable documentation
    boolean isConstructor = false; // if its a contructor instance method
    boolean isStatic = false; // if its a static variable or method
    boolean mFinal = false; // if the method is declared final
    boolean parseLine = true; // continue parsing line
    boolean programEnd = false; // detects that the program is completely parsed
    boolean vFinal = false; // if the variable is declared final


    try {
        reader = new BufferedReader(new FileReader(inFile));
        currentLine = reader.readLine(); // read first line

        while (currentLine != null && !programEnd) {
            theClass.incrementNSourceLines(); // increment number of lines read
            parseLine = true;

            // determine in which scope of the program the parser is currently in
            switch(this.programScope) {
                case 0:  // OUTSIDE OF CLASS
                        //System.out.println("In programScope 0");

                       // first look for class documentation while counting
                       // commented lines
                         if (parseLine && !foundClassDoc &&
                                        parseState.isFullLineComment(currentLine)) {
                           // found the first full line comment
                             foundClassDoc = true;
                           theClass.setHasClassDoc(true);
                           theClass.incrementNCommentLines();
                                                // lines with comments
                           parseLine = false; // go get next line
                         } else if (parseLine && foundClassDoc &&
                                    parseState.isFullLineComment(currentLine)) {
                            theClass.incrementNCommentLines();
                                                // lines with comments
                             parseLine = false;
                         } else if (parseLine && parseState.getInComment()) {
                            // inside of a multiline comment
                             theClass.incrementNCommentLines();
                                                // lines with comments
                           parseState.checkInComment(currentLine);
```

```
                    parseLine = false; // go get next line
                  } else if (parseLine &&
                            parseState.hasInlineComment(currentLine)) {
                      theClass.incrementNCommentLines();
                                        // lines with comments
                    parseLine = true;
                 } else if (parseState.isBlank(currentLine)) {
                      // keeping this check here makes it count a blank
                      // line in middle of multi-line comment as a comment
                      parseLine = false; // go get next line
                  }
                 // look for package statement and class name
                 if (parseLine) {
                     pName = parseState.getPackageName(currentLine);
                     cName = parseState.getClassName(currentLine);
                     if (!pName.equals("none")) {
                           // found the package name
                           theClass.setPackageName(pName);
                          //System.out.println("\tIn package: " + pName);
                           parseLine = false;
                     } else if (!cName.equals("none")) {
                           // found the class name
                           theClass.setClassName(cName);
                          //System.out.println("\tClass: " + cName);
                          this.programScope = 1;
                                // next line will be in the class
                           parseLine = false;
                     }
                  }
                 break;
case 1:  // IN TOP-LEVEL OF CLASS
                 //System.out.println("In programScope 1\n");
                  // look for class/instance variables and methods
                  if (parseLine && parseState.isBlank(currentLine)) {
                      // is a blank line
                      parseLine = false;
                  } else if (parseLine &&
                            parseState.isFullLineComment(currentLine)) {
                      // is a full-line comment
                      theClass.incrementNCommentLines();
                      parseState.setLastContent("comment");
                          // remember comment existence
                      parseLine = false; // go on to next line
                  } else if (parseLine && parseState.getInComment()) {
                      // In the middle of a multi-line comment
                      theClass.incrementNCommentLines();
                          // lines with comments
                    parseState.checkInComment(currentLine);
                          // see if comment continues
                      parseState.setLastContent("comment");
                          // remember comment existence
                      parseLine = false; // go on to next line
                  } else if (parseLine &&
                            parseState.getInVariableDeclaration()) {
                      // In the middle of a multi-line variable declaration
                      parseState.checkInVariableDeclaration(currentLine);
                          // see if continues
                      parseState.setLastContent("notComment");
                  } else if (parseLine &&
                            parseState.getInMethodDeclaration()) {
                      // In the middle of a multi-line method declaration
                      moreParameters =
                          parseState.getMoreParameters(currentLine,
                                                       theClass);
                              // for constructors
                      if (moreParameters != null) {
                          // change the name of the constructor which
                          // was the last instance method
                          // System.out.println("need to add: " +
                          //                    moreParameters);
                          theClass.incrementNMSourceLines(
                              parseState.getMethodType());
                                  // add LOC to method
                          theClass.addMSourceLine(currentLine,
                              parseState.getMethodType());
                      }
                      if (parseState.getIsConstructor() &&
                              moreParameters != null) {
                          theClass.addToMethodName(moreParameters);
                      }
                      // check if still in method declaration
```

```
            parseState.checkInMethodDeclaration(currentLine);
            if (!parseState.getInMethodDeclaration()) {
                parseState.checkInMethodDefinition(currentLine);
                if (parseState.getInMethodDefinition()) {
                    parseState.countMethodBraces(currentLine);
                }
            }
            parseLine = false;
        } else if (parseLine &&
                    parseState.getInMethodDefinition()) {
            // just ended multi-line method decln and is ready
            // to enter the method definition
            programScope = 2; // next line in a method
            // collect info on this line which is the method
            // declaration
            parseState.countMethodBraces(currentLine);
            theClass.incrementNMSourceLines(
                parseState.getMethodType()); // add LOC to method
            theClass.addMSourceLine(currentLine,
                parseState.getMethodType());
        } else if (parseLine &&
                    !parseState.getInMethodDeclaration() &&
                    parseState.isMethodDeclaration(currentLine)) {
            // found beginning of a class/instance meth decln
            if (parseState.isStatic(currentLine)) {
                // Beginning of Class Method declaration
                //System.out.println("Static method.");
                mName = parseState.getMethodName(
                    currentLine,theClass.getClassName());
                mVisibility = parseState.getVisibility(
                    currentLine);
                mFinal = parseState.isFinal(currentLine);
                foundMDoc = parseState.getMethodDoc(currentLine);
                theClass.addClassMethod(mName,mVisibility,
                                        mFinal,foundMDoc);
                parseState.setMethodType("class");
                theClass.incrementNMSourceLines(
                    parseState.getMethodType());
                        // add LOC to method
                // add line content to method
                theClass.addMSourceLine(currentLine,
                    parseState.getMethodType());
            } else { // Beginning of Instance Method declaration
                //System.out.println("Non-static method.");
                mName = parseState.getMethodName(currentLine,
                    theClass.getClassName());
                if (mName.equals(theClass.getClassName())) {
                   // constructor
                   isConstructor = true;
                   mName = mName.concat(
                       parseState.getParameters(currentLine));
                } else {
                   isConstructor = false;
                }
                mVisibility = parseState.getVisibility(
                                        currentLine);
                foundMDoc = parseState.getMethodDoc(currentLine);
                theClass.addInstanceMethod(mName,mVisibility,
                                       foundMDoc,isConstructor);
                parseState.setMethodType("instance");
                theClass.incrementNMSourceLines(
                    parseState.getMethodType());
                        // add LOC to method
                // add line content to method
                theClass.addMSourceLine(currentLine,
                    parseState.getMethodType());
            }
            if (parseState.getInMethodDeclaration()) {
                // come back still in programScope 1
            } else if (parseState.getInMethodDefinition()) {
                programScope = 2; // next line in a method
                // collect info on this line which is the method
                // declaration
                parseState.countMethodBraces(currentLine);
            }
            parseState.setLastContent("notComment");
            parseLine = false;
        } else if (parseState.isProgramEnd(currentLine)) {
            programEnd = true; // saw last closing brace
            if (parseState.hasInlineComment(currentLine)) {
                theClass.incrementNCommentLines();
```

```
                    // lines with comments
                }
            } else if (!parseState.getInMethodDefinition() &&
                    parseState.isVariableDeclaration(
                                currentLine)) {
                // found at least one variable declaration
                if (parseState.isStatic(currentLine)) {
                    //System.out.println("Static variable");
                    vName = parseState.getVariableName(currentLine);
                    vVisibility = parseState.getVisibility(
                                            currentLine);
                    vFinal = parseState.isFinal(currentLine);
                    foundVDoc = parseState.getVariableDoc(
                                           currentLine);
                    if (foundVDoc) {
                        // end-of-line comment -
                        // add to class comment total
                        theClass.incrementNCommentLines();
                            // lines with comments
                      } else if (!foundVDoc &&
                         parseState.getLastContent().equals(
                                                "comment")) {
                        // had a full-line comment on the line
                        // before this - already added
                        // to class comment total
                        foundVDoc = true;
                      }
                      theClass.addClassVariable(vName,vVisibility,
                                             vFinal,foundVDoc);
                } else {
                    //System.out.println("Non-static variable");
                    vName = parseState.getVariableName(currentLine);
                    vVisibility = parseState.getVisibility(
                                            currentLine);
                    foundVDoc = parseState.getVariableDoc(
                                           currentLine);
                    if (foundVDoc) {
                        // end-of-line comment -
                        // add to class comment total
                        theClass.incrementNCommentLines();
                            // lines with comments
                      } else if (!foundVDoc &&
                             parseState.getLastContent().equals(
                                                    "comment")) {
                        // had a full-line comment on the line
                        // before this - already added
                        // to class comment total
                        foundVDoc = true;
                      }
                      theClass.addInstanceVariable(vName,vVisibility,
                                                foundVDoc);
                }
                parseState.setLastContent("notComment");
            }
            break;
case 2:  // IN A METHOD
            //System.out.println("In programScope 2\n");
             theClass.incrementNMSourceLines(
                 parseState.getMethodType()); // add LOC to method
             theClass.addMSourceLine(currentLine,
                 parseState.getMethodType());
                     // add line content to meth
             parseState.countMethodBraces(currentLine);
             if (parseState.getNMLeftBraces() == 1 &&
                 parseState.getNMRightBraces() == 0) {
                     parseState.setInMethodDeclaration(false);
             }
             if (parseState.isMethodEnd()) {
                 // next line will be outside of this method scope
                 programScope = 1;
                 parseLine = true; // want to still count comments
                                   // on that line
             }
             // count comments
             if (parseState.getInComment()) {
                 theClass.incrementNCommentLines();
                     // lines with comments
                 theClass.incrementNMCommentLines(
                     parseState.getMethodType());
               parseState.checkInComment(currentLine);
                     // see if comment continues
```

```
                                parseState.setLastContent("comment");
                                    // remember comment existence
                                //System.out.println("Comment: " + currentLine);
                            } else if (parseState.isFullLineComment(currentLine) ||
                                       parseState.hasInlineComment(currentLine)) {
                                theClass.incrementNCommentLines();
                                    // lines with comments
                                theClass.incrementNMCommentLines(
                                    parseState.getMethodType());
                                parseState.setLastContent("comment");
                                    // remember comment existence
                                //System.out.println("Comment: " + currentLine);
                             }
                            break;
                       default: // should never reach here
                                System.out.println("Not in programScope 0, 1, or 2\n");
                                break;
                    }

                    currentLine = reader.readLine(); // read next line
                }

            } catch (Exception e) {
                System.out.println("Error in parseFile: " + e);
            }

            return this.theClass;
        }

}
```

JStatistics.java

```
/**
 * JStatistics
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JStatistics {

    public JStatistics() {

    }

    private int calculateNDisjointSets(JClass theClass, JCallMatrix theCallMatrix) {
        int nDisjointSets = 0;
        int nMembers = theCallMatrix.getMatrixWidth();
        int nMethods = theCallMatrix.getMatrixHeight();
        int nVariables = nMembers - nMethods;
        int setIndex1 = -1;
        int setIndex2 = -1;
        Vector allSets = new Vector(nMembers); // contains the disjoint sets

        // make each member into an individual set (make-set(member))
        for (int i=0; i<nMembers; i++) {
            Vector set = new Vector();
            set.addElement(theCallMatrix.getUsed(i));
            allSets.addElement(set);
        }

        // find the "edges" and then union the vertices on either side of the edge
        // actually we are traversing the call matrix's rows looking for calls
        // which are the edges
        for (int i=0; i<nMethods; i++) {
            for (int j=0; j<nMembers; j++) {
                if (i != j && theCallMatrix.getUses(i,j) > 0) {
                    // they should be in the same set
                    setIndex1 = findSet(theCallMatrix.getUsed(i), allSets);
                    setIndex2 = findSet(theCallMatrix.getUsed(j), allSets);
                    if (setIndex1 != setIndex2) {
                        // they are not already in the same set
                        allSets = union(setIndex1, setIndex2, allSets);
                    }
```

```
            }
        }
    }

    // print out the elements of the disjoint sets
    /*for (int i=0; i<allSets.size(); i++) {
        Vector s = (Vector) allSets.elementAt(i);
        System.out.println("\nDisjoint set " + (i+1) + ":");
        for (int j=0; j<s.size(); j++) {
            System.out.println("\t" + s.elementAt(j).toString());
        }
    }*/

    // see how many disjoint sets there are
    nDisjointSets = allSets.size();

    return nDisjointSets;
}

private static int findSet(String memberName, Vector allSets) {
    int setNumber = -1;
    boolean foundSet = false;
    Vector set = null;
    String name = null;

    //System.out.println("Looking for set number of " + memberName);

    for (int i=0; i<allSets.size() && !foundSet; i++) {
        set = (Vector) allSets.elementAt(i);
        for (int j=0; j<set.size(); j++) {
            name = set.elementAt(j).toString();
            if (name.equals(memberName)) {
                setNumber = i;
                foundSet = true;
                //System.out.println("Set number: " + setNumber);
            }
        }
    }

    return setNumber;
}

private static Vector union(int index1, int index2, Vector allSets) {
    Vector set1 = null;
    Vector set2 = null;

    // the larger index
    if (index2 > index1) {
        index2 -= 1;
    }

    //System.out.println("Want to union sets " + index1 + " and " + index2);
    set1 = (Vector) allSets.remove(index1);
    set2 = (Vector) allSets.remove(index2);
    set1.addAll(set2);
    allSets.addElement(set1);

    return allSets;
}

public void runStatistics(JClass theClass, JCallMatrix theCallMatrix) {
    System.out.println("\tStarting to run statistics on " +
                       theClass.getClassName() + "...");
    double ratio = 0.0;
    JClassMethod jcm = null;
    JClassVariable jcv = null;
    JInstanceMethod jim = null;
    JInstanceVariable jiv = null;
    int nGlobal = 0;

    /* Number of disjoint sets of class/instance variables and methods */
    int nDisjointSets = this.calculateNDisjointSets(theClass, theCallMatrix);
    theClass.setNDisjointSets(nDisjointSets);

    /* Ratio of class/instance variables and methods that are not private */
    ratio = (theClass.getNNotPrivateMembers()*1.0) /
            (theClass.getNMembers()*1.0);
    theClass.setRNotPrivateMembers(ratio);

    /* Number of global variables */
    for (int i=0; i<theClass.getNClassVariables(); i++) {
```

```
    jcv = theClass.getClassVariable(i);
    if (jcv.getIsGlobal()) {
        nGlobal++;;
    }
}
theClass.setNGlobalVariables(nGlobal);

/* Method line lengths: average and maximum */
//int avgLineLength = 0;
int length = 0;
int maxLineLength = 0;
int totalLength = 0;
Vector lineLengths = null;
for (int i=0; i<theClass.getNClassMethods(); i++) {
    //avgLineLength = 0;
    maxLineLength = 0;
    totalLength = 0;
    jcm = theClass.getClassMethod(i);
    lineLengths = jcm.getLineLengths();
    for (int j=0; j<lineLengths.size(); j++) {
        length = Integer.parseInt(lineLengths.elementAt(j).toString());
        totalLength += length;
        if (length > maxLineLength) {
            maxLineLength = length;
        }
    }
    jcm.setMaxLineLength(maxLineLength);
    jcm.setAvgLineLength(totalLength / lineLengths.size());
}

for (int i=0; i<theClass.getNInstanceMethods(); i++) {
    //avgLineLength = 0;
    maxLineLength = 0;
    totalLength = 0;
    jim = theClass.getInstanceMethod(i);
    lineLengths = jim.getLineLengths();
    for (int j=0; j<lineLengths.size(); j++) {
        length = Integer.parseInt(lineLengths.elementAt(j).toString());
        totalLength += length;
        if (length > maxLineLength) {
            maxLineLength = length;
        }
    }
    jim.setMaxLineLength(maxLineLength);
    jim.setAvgLineLength(totalLength / lineLengths.size());
}

/* Number of method naming policies */
int nCompliant = 0;
int nNotCompliant = 0;
int nPolicies = 0;

// class methods
for (int i=0; i<theClass.getNClassMethods(); i++) {
    jcm = theClass.getClassMethod(i);
    if (jcm.getHasCompliantName()) {
        nCompliant++;
    } else {
        nNotCompliant++;
    }
}
// instance methods
for (int i=0; i<theClass.getNInstanceMethods(); i++) {
     jim = theClass.getInstanceMethod(i);
     if (jim.getHasCompliantName()) {
         nCompliant++;
     } else {
         nNotCompliant++;
     }
}
if (nNotCompliant == 0 || nCompliant == 0) {
    nPolicies = 1;
} else {
    nPolicies = 2;
}
theClass.setNMethNamingPolicies(nPolicies);

/* Number of variable naming policies */
nCompliant = 0;
nNotCompliant = 0;
nPolicies = 0;
```

```
        // class variables
        for (int i=0; i<theClass.getNClassVariables(); i++) {
            jcv = theClass.getClassVariable(i);
            if (jcv.getHasCompliantName()) {
                nCompliant++;
            } else {
                nNotCompliant++;
            }
        }
        // instance variables
        for (int i=0; i<theClass.getNInstanceVariables(); i++) {
            jiv = theClass.getInstanceVariable(i);
            if (jiv.getHasCompliantName()) {
                nCompliant++;
            } else {
                nNotCompliant++;
            }
        }
        if (nNotCompliant == 0 || nCompliant == 0) {
            nPolicies = 1;
        } else {
            nPolicies = 2;
        }
        theClass.setNVarNamingPolicies(nPolicies);

        /* Comment density for class */
        ratio = (theClass.getNCommentLines()*1.0) / (theClass.getNSourceLines()*1.0);
        theClass.setRComments(ratio);

        /* Comment density per method */
        for (int i=0; i<theClass.getNClassMethods(); i++) {
            jcm = theClass.getClassMethod(i);
            ratio = (jcm.getNCommentLines()*1.0) / (jcm.getNSourceLines()*1.0);
            jcm.setRComments(ratio);
        }
        for (int i=0; i<theClass.getNInstanceMethods(); i++) {
            jim = theClass.getInstanceMethod(i);
            ratio = (jim.getNCommentLines()*1.0) / (jim.getNSourceLines()*1.0);
            jim.setRComments(ratio);
        }
    }

}
```

JVariable.java

```
/**
 * JVariable
 *
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JVariable {
    /* Instance Variables */
    protected Vector criticism;
    protected double grade;
    protected boolean hasCompliantName; // if it has compliant name
    protected String label; // "IV#" or "CV#", # is from 1 to n
    protected int nCallee;
    protected String variableName; // the variable name
    protected boolean variableDoc; // variable comment
    protected String variableType; // "class", "instance", or null
    protected String visibility; // public,protected,private,default

    /* Public Methods */

    /**
     * Class constructor
     */
    protected JVariable() {
        this.criticism = new Vector(5);
        this.grade = 100.0;
        this.hasCompliantName = false;
```

```
        this.label = null;
        this.nCallee = 0;
        this.variableName = null;
       this.variableDoc = false; // assume no variable comment
        this.variableType = null;
        this.visibility = "default"; // default (package) visibility by default
    }

    public void addCriticism(String criticism) {
        this.criticism.addElement(criticism);
    }

    public Vector getCriticism() {
        return this.criticism;
    }

    public double getGrade() {
        return this.grade;
    }

    public boolean getHasCompliantName() {
        return this.hasCompliantName;
    }

    public String getLabel() {
        return this.label;
    }

    public int getNCallee() {
        return this.nCallee;
    }

    public String getVariableName() {
        return this.variableName;
    }

    public String getVariableType() {
        return this.variableType;
    }

    public String getVisibility() {
        return this.visibility;
    }

    protected boolean hasVariableDoc(){
        return this.variableDoc;
    }

    public void setGrade(double grade) {
        this.grade = grade;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public void setNCallee(int nCallsTo) {
        this.nCallee = nCallsTo;
    }

    public String toString() {
        return("Summary for: " + this.getVariableName() + " (" + this.getLabel() +
               ")\n" +
               "\tVariable visibility: " + this.getVisibility() + "\n" +
               "\tHas variable documentation: " + this.hasVariableDoc() + "\n" +
               "\tNumber of call to this: " + this.getNCallee() + "\n" +
               "\tHas compliant name: " + this.getHasCompliantName() + "\n" +
               "\tGrade: " + this.getGrade());
    }

}
```

JVisualizer.java

```
/**
 * JVisualizer
 *
```

```
 * @author     Andrea Goethals
 * @version    %I% %G%
 */

import java.util.*;

public class JVisualizer {

    /* Instance Variables */
    private JParser classParser; // parser of the java source code
    private JCritique critique;
    private JModeler model;
    private JStatistics statistics;
    private JClass theClass; // result of classParser's parsing
    private JCallMatrix theCallMatrix; // a call/use matrix to measure class cohesion

    /* Public Methods */

    /**
     * Class constructor
     */
    public JVisualizer() {
        this.classParser = new JParser();
        this.critique = null;
        this.model = null;
        this.statistics = null;
        this.theClass = null;
        this.theCallMatrix = null;
    }

    /**
     * Driver
     */
    public static void main(String[] args) {
        Calendar startTime, endTime;
        Date startTimeMillis, endTimeMillis;
        long runTime;
        TimeZone zone = TimeZone.getTimeZone("EST");

        if (args.length != 1) {
            System.out.println("\n Usage: java JVisualizer <yourfile.java>\n");
           return;
        } else {
            printAppGreeting(args[0]);
        }

        JVisualizer visualizer = new JVisualizer();

        /* Start timing the runtime */
        startTime = Calendar.getInstance(zone);
        startTimeMillis = new Date();
        System.out.println(" Time started:\t" + startTime.getTime() + "\n");

        /* parse file: phase 1 */
        visualizer.theClass = visualizer.classParser.parseFile(args[0]);

        /* build call/use matrix: phase 2 */
        visualizer.theCallMatrix = new JCallMatrix();
        visualizer.theCallMatrix =
            visualizer.theCallMatrix.buildMatrix(visualizer.theClass);

        /* run all the statistics on the class: phase 3 */
        visualizer.statistics = new JStatistics();
        visualizer.statistics.runStatistics(visualizer.theClass,
            visualizer.theCallMatrix);

        /* critique the class: phase 4 */
        visualizer.critique = new JCritique();
        visualizer.critique.doCritique(visualizer.theClass);

        /* draw the class: phase 5 */
        visualizer.model =
            new JModeler(visualizer.theClass, visualizer.theCallMatrix);
        visualizer.model.drawModel();

        /* Stop timing the runtime */
        endTime = Calendar.getInstance(zone);
        endTimeMillis = new Date();
        System.out.println("\n Time finished:\t\t\t" + endTime.getTime());
        runTime = endTimeMillis.getTime() - startTimeMillis.getTime();
        System.out.println(" Runtime in milliseconds:\t" + runTime);
```

```
        System.out.println(" Runtime in seconds:\t\t" + runTime/1000 + "\n");
    }

    /**
     * Prints the program's name
     *
     * @param         file    Java source file (*.java)
     */
    private static void printAppGreeting(String file) {
        System.out.println("\n ***************************************");
        System.out.println(" *                                     *");
        System.out.println(" *    Welcome to JVisualizer ver 0.1   *");
        System.out.println(" *                                     *");
        System.out.println(" ***************************************");
        System.out.println("\n Program to visualize: " + file + "\n");
    }

}
```

Makefile

```
#
# Makefile to build JVisualizer
#
# Andrea Goethals
#

#########################################################
# Macro and variable declarations
#########################################################

# The class files
CLASSES = JCallMatrix.class \
               JClass.class \
               JClassMethod.class \
               JClassVariable.class \
               JColorConstants.class \
               JCritique.class \
               JGrammar.class \
               JInstanceMethod.class \
               JInstanceVariable.class \
               JListener.class \
               JMethod.class \
               JModeler.class \
               JParser.class \
               JParseState.class \
               JStatistics.class \
               JVariable.class \
                JVisualizer.class


#########################################################
# Includes
#########################################################

# Include the configuration file
include make.cfg

#########################################################
# Targets
#########################################################

# The all target
all: $(CLASSES)

# Clean up
clean:
        rm -f $(CLASSES)
```

---

make.cfg

```
#
# make.cfg
#
```

```
# Configuration file for makefiles
#

#########################################################
# Macro and variable declarations
#########################################################

# Update suffixes
.SUFFIXES: .java .class

#########################################################
# Rules
#########################################################

# The rule to build a class file
.java.class:
        javac $<
```

---

test

```
#!/bin/bash

javac JVisualizer.java

if [ "$#" -ne 1 ]; then
    java JVisualizer inputprograms/Grammar.java
else
    java JVisualizer $1
fi
```

APPENDIX B
JVISUALIZER DEFAULT CONFIGURATION FILES

The contents of the file "customGrading" is not listed here because it is the same as the file "defaultGrading" which is included below. A user could edit the file "customGrading" to override the default grading parameters. Similarly, the contents of the file "customMetrics" is not included here because it is the same as the file "defaultMetrics" which is included below.

defaultGrading

```
# These are the default grading policies used by JVisualizer
#
# Author: Andrea Goethals
# Last modified: 02-10-01
#
# Notes: Make sure that the penalty is a negative float if you
#           want it to act as a penalty.
#       To "delete" a guideline make the penalty 0.

# Penalty for having more than the
# maximum acceptable number of disjoint sets
MORE_C_INT_MAX-NUM-DISJOINT-SETS=-15.0

# Penalty for having more than the
# maximum acceptable number of public class/instance variables
# and methods
MORE_C_INT_MAX-NUM-PUB-MEMS=-5.0

# Penalty for having more than the
# maximum acceptable ratio of public, protected and/or
# default class/instance variables and methods to all
# class/instance variables and methods
MORE_C_FLOAT_MAX-RAT-NONPRIV-MEMS=-15.0

# Penalty for having more than the
# maximum acceptable number of public class variables that
# are not declared final
MORE_C_INT_MAX-NUM-GLOBAL-VARS=-10.0

# Penalty for having more than the
# maximum acceptable number of lines of code per class
MORE_C_INT_MAX-NUM-LOC=-15.0

# Penalty for having more than the
# maximum acceptable number of class/instance method naming policies
MORE_C_INT_MAX-NUM-METH-NAME-POLS=-5.0

# Penalty for having more than the
# maximum acceptable number of class/instance variable naming policies
MORE_C_INT_MAX-NUM-VAR-NAME-POLS=-5.0

# Penalty for having less than the
# minimum acceptable ratio of class comment density
LESS_C_FLOAT_MIN-RAT-COM-DENSITY=-10.0
```

```
# Penalty for not having
# class documentation
NOT_C_BOOL_HAS-DOC=-20.0

# Penalty for not having
# compliant class name
NOT_C_BOOL_IS-NAME-COMP=-30.0

# Penalty for having more than the
# maximum acceptable number of lines of code per method
MORE_M_INT_MAX-NUM-LOC=-15.0

# Penalty for having more than the
# maximum acceptable maximum number of characters per line per method
MORE_M_INT_MAX-MAX-LINE-LENGTH=-10.0

# Penalty for having more than the
# maximum acceptable average number of characters per line per method
MORE_M_INT_MAX-AVG-LINE-LENGTH=-15.0

# Penalty for having less than the
# minimum acceptable ratio of method comment density
LESS_M_FLOAT_MIN-RAT-COM-DENSITY=-5.0

# Penalty for not having
# class/instance method documentation
NOT_M_BOOL_HAS-DOC=-15.0

# Penalty for not having
# compliant class/instance method name
NOT_M_BOOL_IS-NAME-COMP=-30.0

# Penalty for not having
# class/instance variable documentation
NOT_V_BOOL_HAS-DOC=-20.0

# Penalty for not having
# compliant class/instance variable name
NOT_V_BOOL_IS-NAME-COMP=-30.0
```

---

defaultMetrics

```
# These are the default metrics properties used by JVisualizer
# to evaluate a Java program
#
# Author: Andrea Goethals
# Last modified: 02-10-01

# maximum acceptable number of disjoint sets
C_INT_MAX-NUM-DISJOINT-SETS=1

# maximum acceptable number of public class/instance variables
# and methods
C_INT_MAX-NUM-PUB-MEMS=15

# maximum acceptable ratio of public, protected and/or
# default class/instance variables and methods to all
# class/instance variables and methods
C_FLOAT_MAX-RAT-NONPRIV-MEMS=0.5

# maximum acceptable number of public class variables that
# are not declared final
C_INT_MAX-NUM-GLOBAL-VARS=0

# maximum acceptable number of lines of code per class
C_INT_MAX-NUM-LOC=500

# maximum acceptable number of class/instance method naming policies
C_INT_MAX-NUM-METH-NAME-POLS=1

# maximum acceptable number of class/instance variable naming policies
C_INT_MAX-NUM-VAR-NAME-POLS=1

# minimum acceptable ratio of class comment density
C_FLOAT_MIN-RAT-COM-DENSITY=0.25
```

```
# existence of class documentation
C_BOOL_HAS-DOC=true

# compliant class name
C_BOOL_IS-NAME-COMP=true

# maximum acceptable number of lines of code per method
M_INT_MAX-NUM-LOC=45

# maximum acceptable maximum number of characters per line per method
M_INT_MAX-MAX-LINE-LENGTH=80

# maximum acceptable average number of characters per line per method
M_INT_MAX-AVG-LINE-LENGTH=30

# minimum acceptable ratio of method comment density
M_FLOAT_MIN-RAT-COM-DENSITY=0.05

# existence of class/instance method documentation
M_BOOL_HAS-DOC=true

# compliant class/instance method name
M_BOOL_IS-NAME-COMP=true

# existence of class/instance variable documentation
V_BOOL_HAS-DOC=true

# compliant class/instance variable name
V_BOOL_IS-NAME-COMP=true
```

APPENDIX C
VISUALIZATION SCREEN CAPTURES

This appendice contains screen captures of the visualizations created by JVisualizer for all thirteen input programs. To save space, all of the graphics have been cropped and scaled down, which accounts for the blurriness of some of these pictures. Figure A-1 shows the color legend that appears on JVisualizer's GUI to assist in determining the grades given to each class and member of the class.

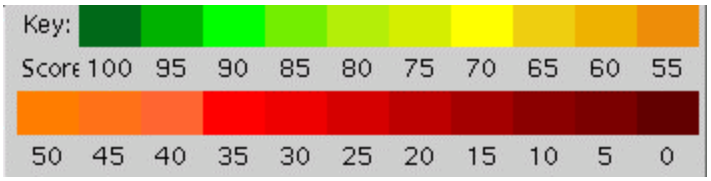

Figure C-1: The color legend on JVisualizer's GUI

Notes: Each color in the legend has a corresponding grade labeled below the color

1. BigCube.java

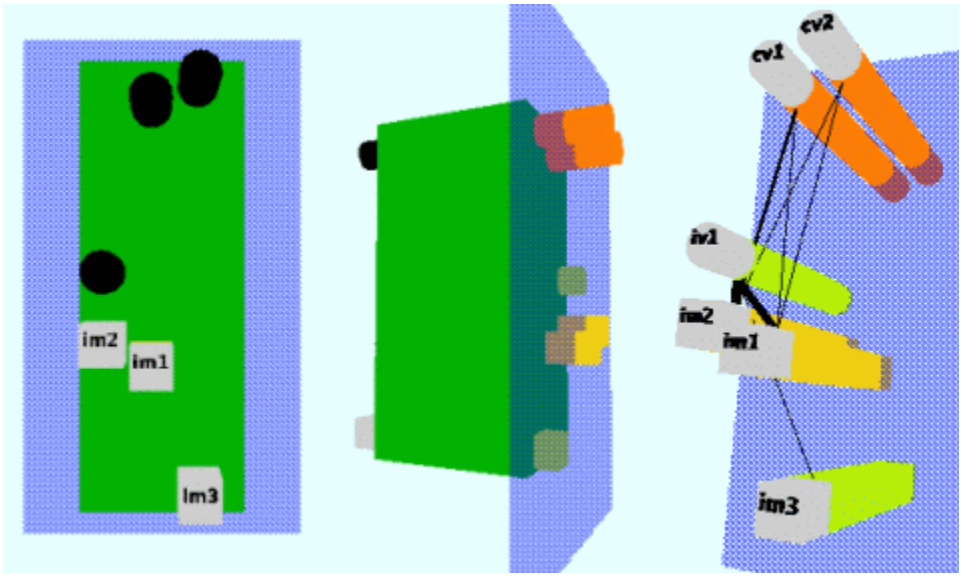

Figure C-2: BigCube.java as seen from the front (from any class), back (from any class) and from within the class

2. Grammar.java

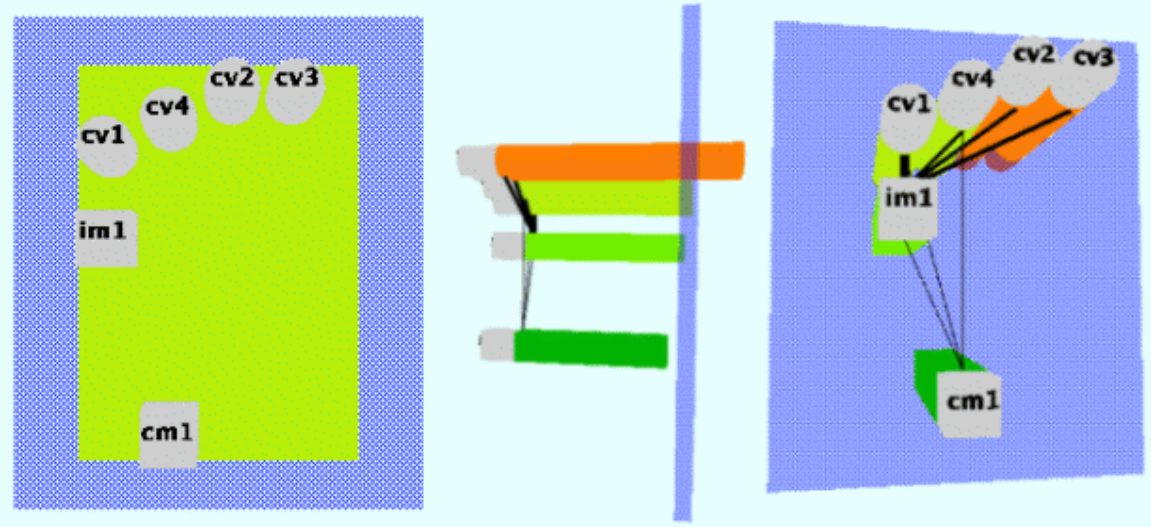

Figure C-3: Grammar.java as seen from the front (from any class), side (from within the class) and front (from within the class)

3. HazardChecker.java

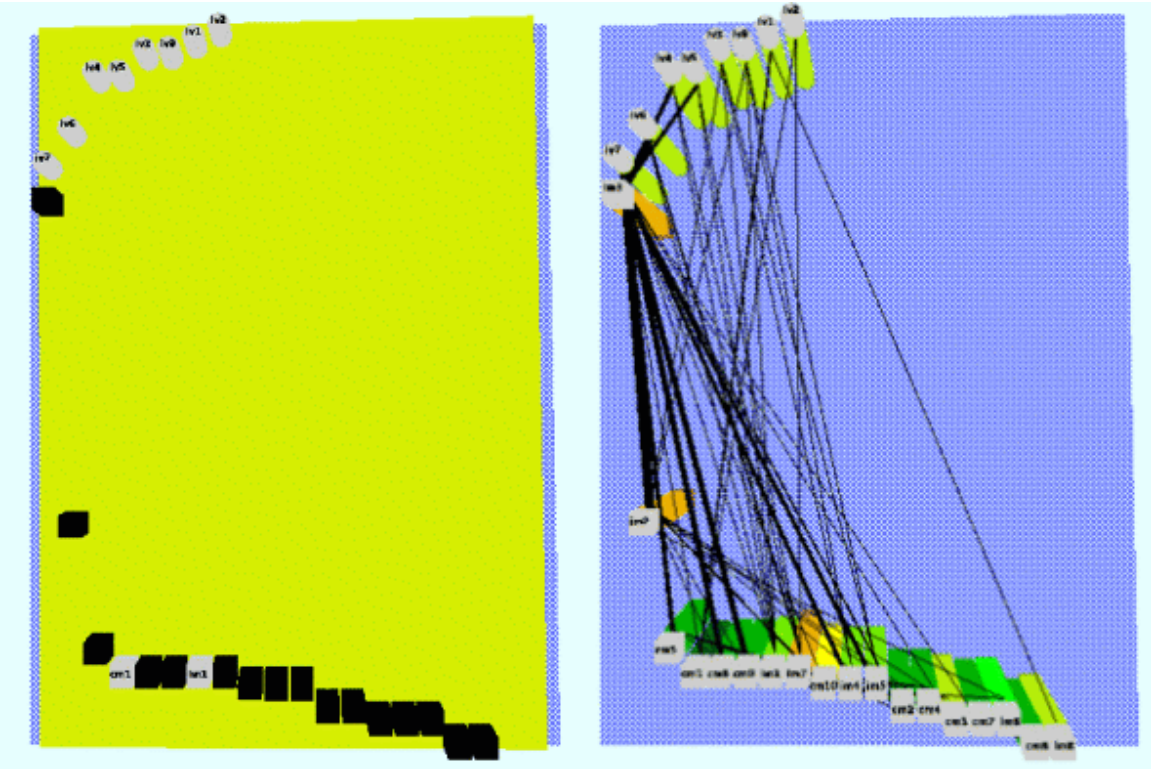

Figure C-4: HazardChecker.java as seen from the front (from any class), and front (from within the class)

4. Hazard.java

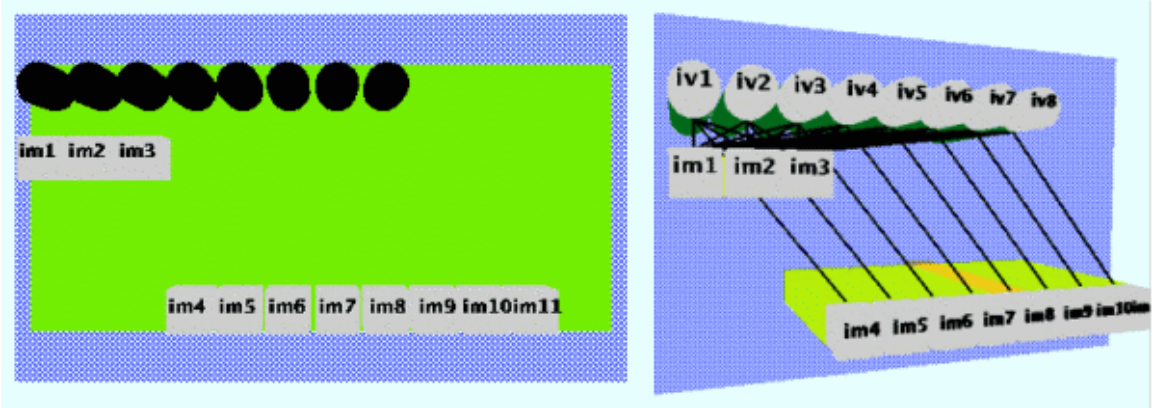

Figure C-5: Hazard.java as seen from the front (from any class), and front (from within the class)

5. HtmlLink.java

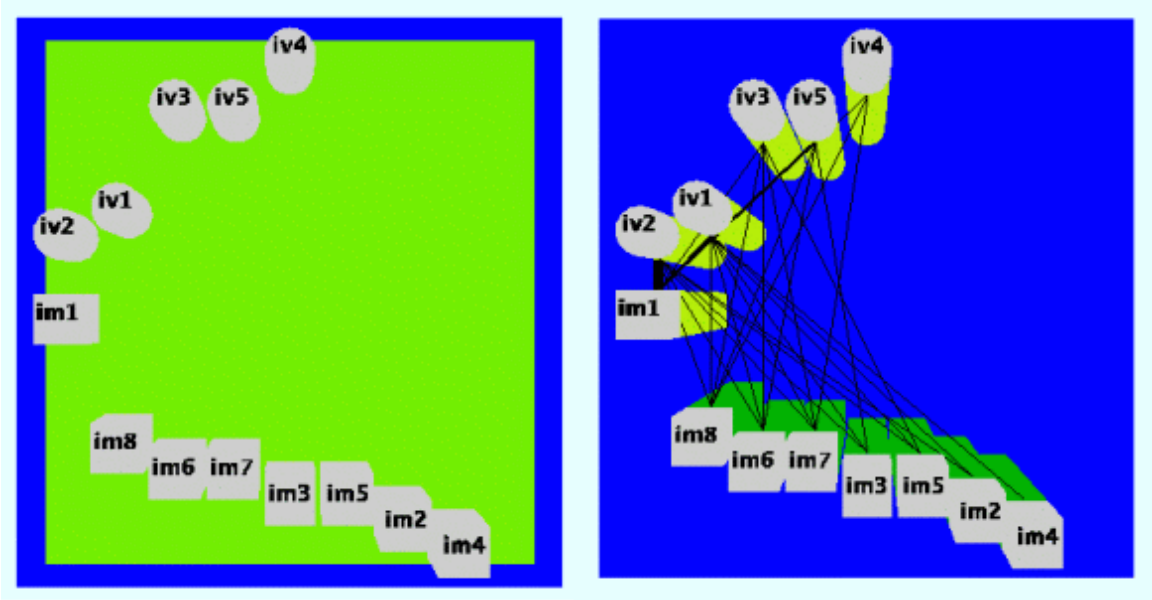

Figure C-6: HtmlLink.java as seen from the front (from any class), and front (from within the class)

6. IWonEvent.java

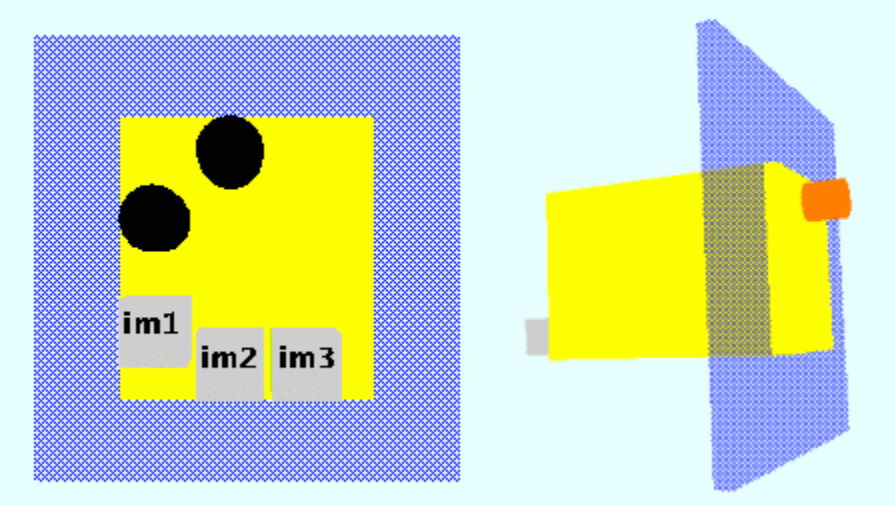

Figure C-7: IWonEvent.java as seen from the front (from any class), and back (from any class)

7. NapsterSong.java

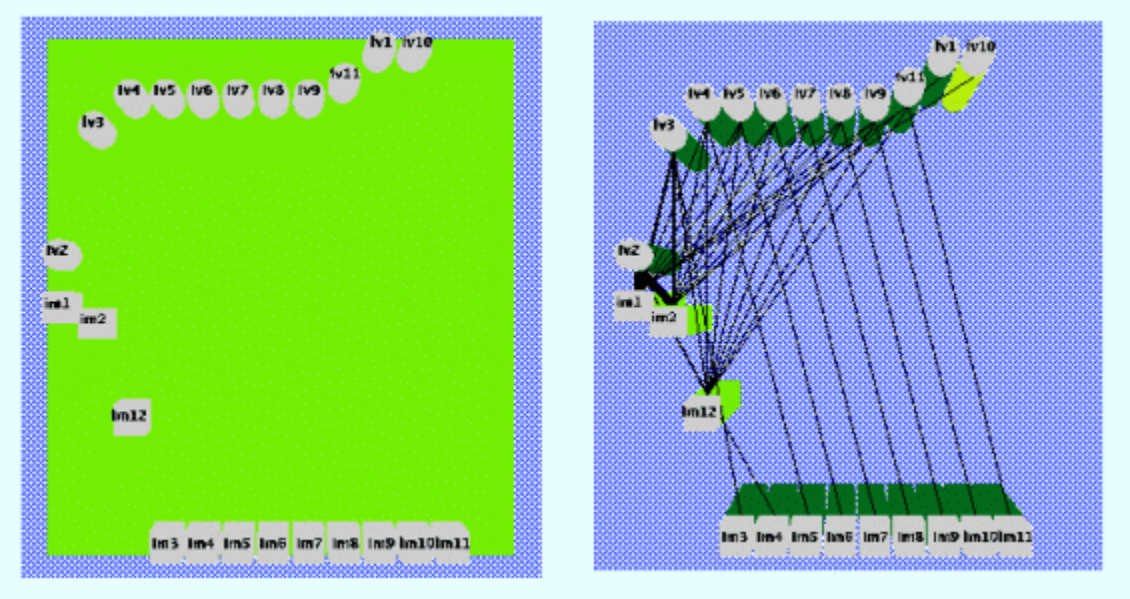

Figure C-8: NapsterSong.java as seen from the front (from any class), and front (from within the class)

8. NotaryPublic.java

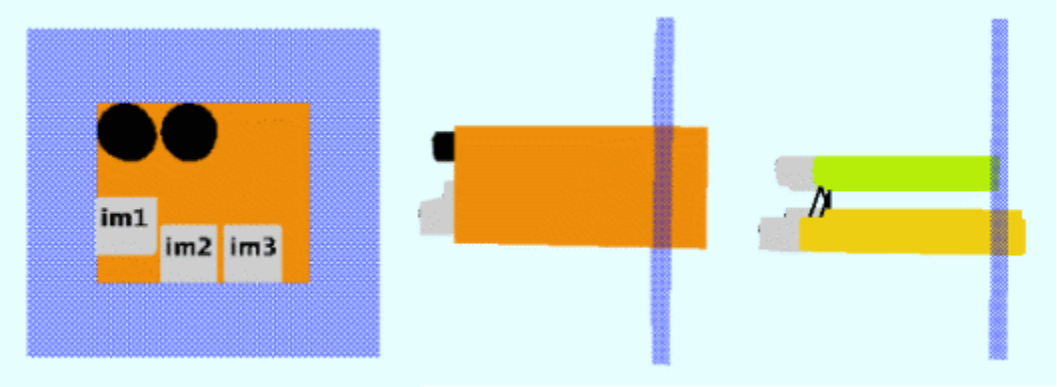

Figure C-9: NotaryPublic.java as seen from the front (from any class), side (from any class), and side (from within the class)

9. Scanner.java

Figure C-10 (Top of Figure is to the right): Scanner.java as seen from the front (from any class), and front (from within the class)

10. ScoreBoardSimulator.java

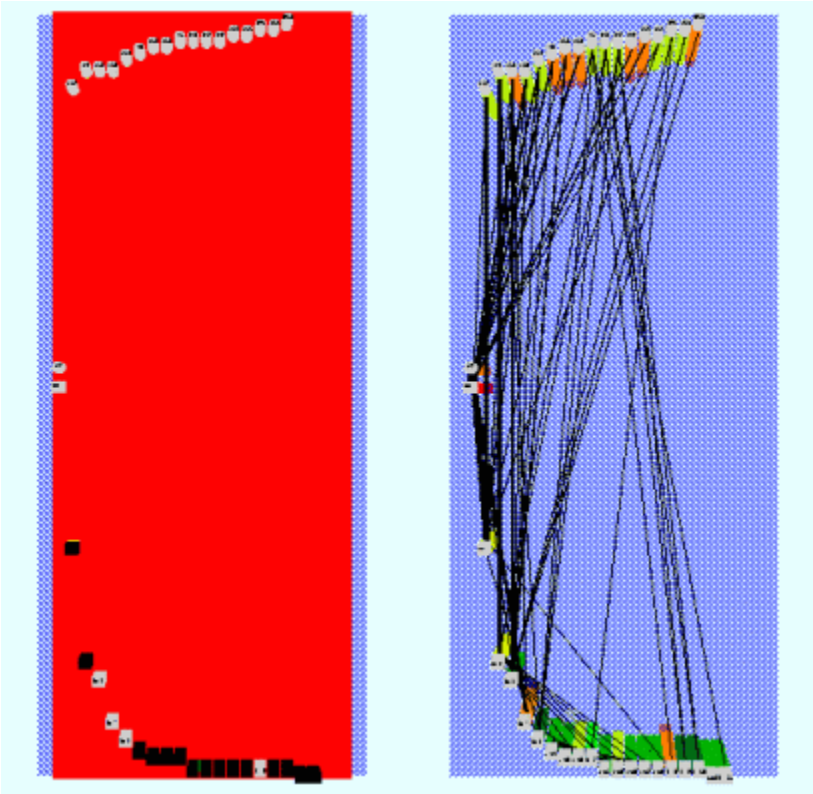

Figure C-11: ScoreBoardSimulator.java as seen from the front (from any class), and front (from within the class)

11. StripQualifiers.java

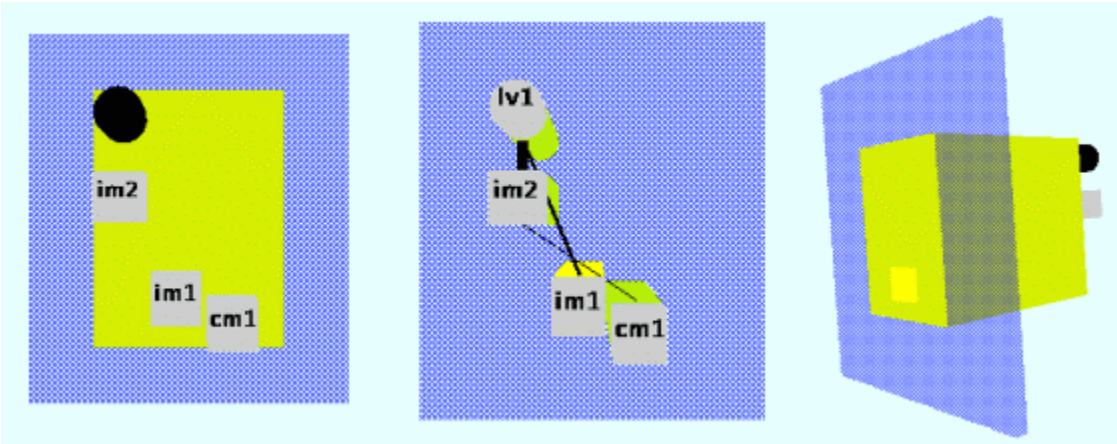

Figure C-12: StripQualifiers.java as seen from the front (from any class), front (from within the class), and back (from any class)

12. TinyParser.java

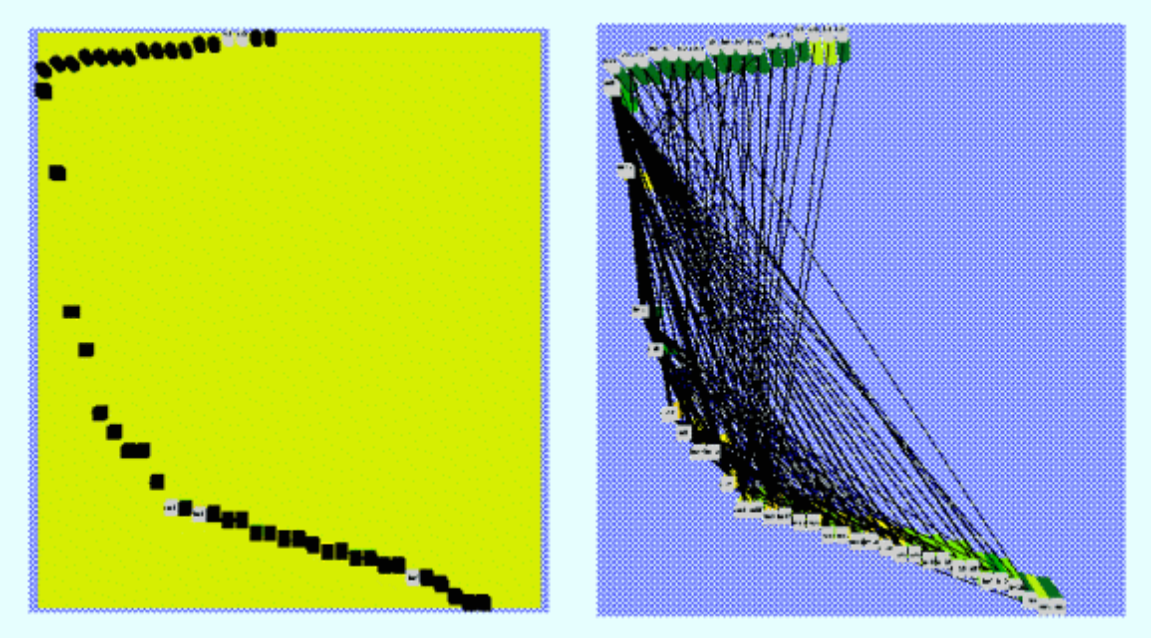

Figure C-13: TinyParser.java as seen from the front (from any class), and front (from within the class)

13. website.java

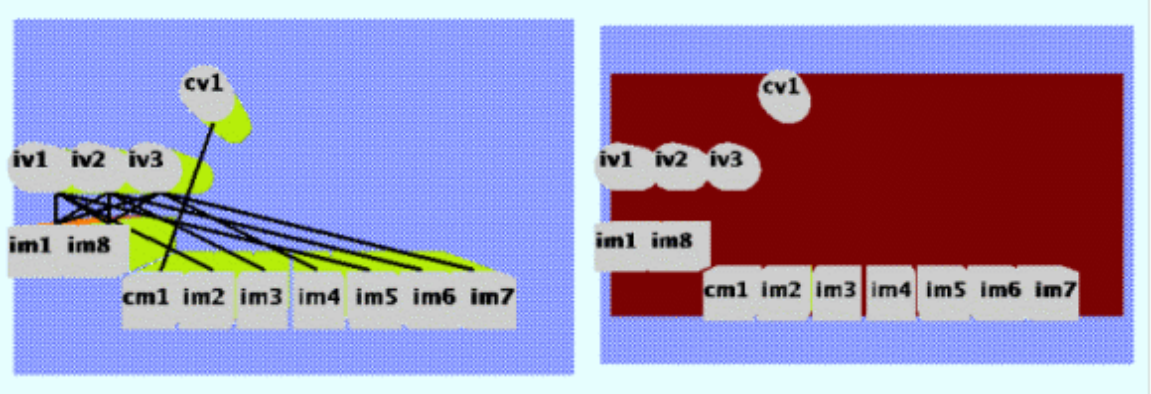

Figure C-14: website.java as seen from the front (from within the class), and front (from any class)

REFERENCES

(Ambler 2000) Scott W. Ambler. "Writing Robust Java Code: The AmbySoft Inc. Coding Standards for Java v17.01d," January 2000. (http://www.ambysoft.com/javaCodingStandards.html)

(Attali et al. 1998) Isabelle Attali, Denis Caromel, and Marjorie Russo. "A Formal Executable Semantics for Java," *Proceedings of Formal Underpinnings of Java - An OOPSLA'98 Workshop*, Vancouver, October 1998.

(Attali et al. 2001a) Isabelle Attali, Denis Caromel, and Marjorie Russo. "Graphical Visualization of Java Objects, Threads, and Locks," *IEEE Distributed Systems Online*, Vol. 2, No. 1, January 2001.

(Attali et al. 2001b) Isabelle Attali, Denis Caromel, and Marjorie Russo. "Jitan vs. Other Tools," *IEEE Distributed Systems Online*, Vol. 2, No. 1, January 2001.

(Badeaux 1999) Christie Badeaux. "Netscape's Software Coding Standards Guide for Java," On-line Publication, Netscape Communications Corporation, 1999. (http://developer.netscape.com/docs/technote/java/codestyle.html)

(Bassil and Keller, 2001) Sarita Bassil and Rudolf K. Keller. "Software Visualization Tools: Survey and Analysis," In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'2001)*, Toronto, Ontario, Canada, May 2001..

(Catton and Murphy 2001) Andrew Catton and Gail C. Murphy. "Scaling Dynamic Architectural Software Visualizations," Proceeding of the International Conference on Software Engineering, Workshop on Software Visualization, Toronto, 2001.

(Cheng 1998) May Cheng. "SV Techniques: Animation, 3D, Color, Sound, UI," Lecture, Software Visualization Course, Georgia Institute of Technology, Atlanta, January 1998.

(Chidamber and Kemerer 1994) Shyam R. Chidamber, and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994.

(Coad and Mayfield 1999) Peter Coad and Mark Mayfield. *Java Design: Building Better Apps & Applets*, Second Edition, Yourdon Press, Upper Saddle River, NJ, 1999.

(Daconta et al. 2000) Michael C. Daconta, Eric Monk, J. Paul Keller, and Keith

Bohnenberger. *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs*,. Wiley Computing Publishing, New York, NY, 2000.

(De Pauw et al. 1993) W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. "Visualizing the Behavior of Object-Oriented Systems," *OOPSLA '93 Conference Proceedings*, Washington, DC, September 1993, pp. 326-337.

(De Pauw et al. 1994) W. De Pauw, D. Kimelman, and J. Vlissides. "Modeling Object-Oriented Program Execution", ECOOP '94, July 1994, Bologna, Italy, in *Lecture Notes in Computer Science*, Vol. 821, Springer Verlag, 163-182.

(Eckel 1998) Bruce Eckel, *Thinking in Java*. Prentice Hall PTR, Upper Saddle River, NJ, 1998.

(Eckel 2000) Bruce Eckel, "Java Programming Guidelines," *Thinking in Java*. Second Edition, Revision 11, Appendix C, Prentice Hall PTR: Upper Saddle River, NJ; 2000.

(Edsall 1999) Robert M. Edsall. "General principles of color use in visualization applications," Unit 47: On-Screen Visualization, The NCGIA Core Curriculum for Technical Programs; March 1999.

(Eick and Karr, 2000) Stephen G. Eick and Alan F. Karr, "Visual Scalability," IEEE Transactions on Visualization and Computer Graphics, June, 2000

(Eick et al. 2000) Stephen G. Eick, Paul Schuster, Audris Mockus, Todd L. Graves and Alan F. Karr, "Visualizing Sofware Changes," Technical Report, National Institute of Statistical Sciences, December, 2000.

(Eisenstadt et al. 1993) M. Eisenstadt, B. Price. and J. Domingue. "Software Visualization as a Pedagogical Tool," *Instructional Science*, Vol. 21, 1993.

(Elish 2001) Mahmoud Elish. "An Experiment Evaluating the Conformance of Open Source Java Programs to Standard Coding Practices," Student paper for a course on Software Engineering Experimentation, George Mason University, Fairfax, VA, 2001.

(Feigenbaum 2000) Barry A. Feigenbaum. "Java Coding Guidelines," Draft first edition, On-line essay, March 2000. (http://www.cs.utexas.edu/users/barryf/guidedocX.html)

(Flood and Carson 1993) Robert L. Flood and Ewart R. Carson. *Dealing with Complexity: An Introduction to the Theory and Application of Systems Science*, Second Edition, Plenum Press, New York, NY, 1993.

(Foley et al. 1997) James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*, Second Edition in C, Addison-Wesley Publishing Company, Reading, MA, 1997.

(Friedman 2001) Erich Friedman. “Spheres in Cube,” Webpage, 2001. (http://www.stetson.edu/~efriedma/sphincub/)

(Fussell 1997) Mark L. Fussell. "Java Development Standards," Version 0.7, ChiMu Corporation, Sunnyvale, CA, 1997.

(Fussell 1998) Mark L. Fussell. "ChiMu OO and Java Development: Guidelines and Resources," Version 1.8, Sunnyvale, CA, 1998.

(Graham and Kennedy 2001) M. Graham, and J. B. Kennedy, "Combining Linking and Focusing Techniques for a Multiple Hierarchy Visualisation," in *Proc. IV 2001*, pp. 425-432, IEEE Computer Society Press, London, UK, July 25-27, 2001.

(Gosling et al. 2000) James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. "The Java Language Specification," Second Edition, Sun Microsystems, Inc., Santa Clara, CA, 2000.

(GSS 2001) Geotechnical Software Services. "Java Programming Style Guidelines," Version 2.9, Stavanger, Norway, February 2001.

(Haahr 1999) Paul Haar. "A Programming Style for Java," On-line Essay, October 1999. (http://www.webcom.com/~haahr/essays/java-style/)

(Haibt 1959) L. M. Haibt. “A Program to Draw Multi-Level Flowcharts”, In *Proceedings of the Western Joint Computer Conference*, Vol. 15, San Francisco, 1959, pp. 131-137.

(Harisan 2001) Harisan Communications. “Charts Software,” Webpage, 2001. (http://www.harisan.com/Services/Applications/Charts_Software/Graph3D3.html).

(Henderson-Sellers 1996) Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall Inc., Upper Saddle River, NJ, 1996.

(Hitz and Montazeri 1996) Martin Hitz, and Behzad Montazeri. "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE Transactions on Software Engineering*, Vol. 22, No. 4, April 1996.

(Hubbard 2001) John R. Hubbard. *Data Structures with Java*, Schaum's Outline Series, McGraw-Hill,: New York, NY, 2001.

(IBM 1996) IBM Almaden Research Center. "IBM Program Visualizer (PV): Tutorial and Reference Manual," Version 0.8.3, IBM Almaden Research Center, Silicon Valley, CA, June 1996.

(Iseran 2001) Iseran. "Iseran Java Style Guidelines," On-line Essay, 2001. (http://www.iseran.com/Win32/Articles/java_style_guidelines.html)

(King et al. 1999) Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel. "Code Conventions for the JavaTM Programming Language," Sun Microsystems, Inc., Santa Clara, CA, Revised April 20, 1999.

(Kraemer et al. 2001) Eileen Kraemer, Mihail Tudoreanu, and Ashley Taylor. "Why Johnny Won't Visualize," *Proceedings of the International Conference on Software Engineering: Workshop on Software Visualization*, Toronto, 13-14 May 2001.

(Lea 2000) Doug Lea. "Draft Java Coding Standard," State University of New York at Oswego, Oswego, NY, August 1997, Revised February 2000.

(Loeffler 2000) David Loeffler. "Java Coding Guidelines," Draft revision 1.0, On-line report, June 2000. (http://www.onr.com/user/loeffler/java/jguide.html)

(Lorenz and Kidd 1994) Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics*. Prentice-Hall Inc., Englewood Cliffs, NJ; 1994.

(Maletic et al. 2001) Jonathan I. Maletic, Jason Leigh, and Andrian Marcus. "Visualizing Software in an Immersive Virtual Reality Environment," *Proceedings of the International Conference on Software Engineering: Workshop on Software Visualization*, Toronto, 13-14 May 2001.

(Mitchell 2001) Kenneth J Mitchell. “Three Dimensional Widgets”, Webpage, 2001. (http://www.dcs.napier.ac.uk/vrprj/tdw/tdw.html)

(MMS 2001) "Getting Started With JStyleTM", Man Machine Systems, Chennai, India, June 2001.

(Neal et al. 1997a) Ralph D. Neal, H. Roland Weistroffer, and Richard J. Coppins. "An Improved Suite of Object Oriented Software Measures," NASA/WVU Software Research Lab, Fairmont, WV, 1997.

(Neal et al. 1997b) Ralph D. Neal, H. Roland Weistroffer, and Richard J. Coppins. "A Taxonomy of Object-Oriented Measures Modeling the Object-Oriented Space," Technical Report, NASA, 1997.

(Parker et al. 2000) Greg Parker, Glenn Franck and Colin Ware. "Visualization of Large Nested Graphs in 3D: Navigation and Interaction," Journal of Visual Languages and Computing, Vol. 9, pp. 299-317, July 2000.

(Petre et al., 1998) Marian Petre, Alan Blackwell and Thomas Green. "Cognitive Questions in Software Visualization," *Software Visualization: Programming as a Multimedia Experience*, Chapter 30, MIT Press, Cambridge, MA, 1998.

(Price et al. 1993) B. A. Price, R. M. Baecker, and I. S. Small. "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, Vol. 4, No. 3,

pp. 211-266; 1993.

(Rosenberg 1998) Linda H. Rosenberg. "Applying and Interpreting Object Oriented Metrics," Presentation at the Software Technology Conference, Salt Lake City, Utah, April 1998.

(Sommerville 2001) Ian Sommerville. *Software Engineering*. Sixth Edition, Addison-Wesley Publishers, Harlow, England, 2001.

(Spence 2001) Robert Spence. *Information Visualization*, ACM Press, Harlow, England, 2001.

(Stasko 1992) John T. Stasko. "Three-Dimensional Computation Visualization," Technical Report GIT-GVU-92-20, Georgia Institute of Technology, Atlanta, 1992.

(Stasko et al. 1998) John Stasko, John Domingue, Marc H. Brown and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

(Storey 2001) Margaret-Anne Storey. "SHriMP Views: An Interactive Environment for Exploring Java Programs," P*roceedings of the International Conference on Software Engineering: Workshop on Software Visualization*, Toronto, 13-14 May 2001.

(Storey et al. 1998) M.-A.D. Storey, K. Wong, and H.A. Müller. "How Do Program Understanding Tools Affect How Programmers Understand Programs?" Presented at WCRE '97, Amsterdam, Holland, 1997.

(Sun, 2000) The Java 3D API, Version 1.2, Sun Microsystems, Inc., Santa Clara, CA, March 2000.

(Sun,2001) The Java 2 Platform, Standard Edition, Version 1.3.1 API,. Sun Microsystems, Inc., Santa Clara, CA, 2001.

(Systä et al. 2000) Tarja Systä, Ping Yu, and Hausi Müller. "Analyzing Java Software by Combining Metrics and Program Visualization," *Proceedings of CSMR Conference*, Zurich, February 2000.

(Tyma et al. 1996) Paul M. Tyma, Gabriel Torok, and Troy Downing. *Java Primer Plus*, Waite Group Press, Corte Madera, CA, 1996

(Venners 1998a) Bill Venners. "Designing Fields and Methods: How to Keep Fields Focused and Methods Decoupled," *JavaWorld*, On-line Magazine, March 1998. (http://www.javaworld.com/javaworld/jw-04-1998/jw-04-techniques.html)

(Venners 1998b) Bill Venners. "What's a Method to Do?: How to Maximize Cohesion While Avoiding Explosion," *JavaWorld,* On-line Magazine, April 1998.

(http://www.javaworld.com/javaworld/jw-05-1998/jw-05-techniques.html)

(Venners 1999) Bill Venners. "Design with Static Members: How to Put Static Fields and Methods to Work," *JavaWorld,* On-line Magazine, March 1999. (http://www.javaworld.com/javaworld/jw-03-1999/jw-03-techniques.html)

(Venners 2002) Bill Venners. *API Design: Best Practices in Object-Oriented API Design in Java*, On-line book, Artima.com, January 2002. (http://www.artima.com/interfacedesign/contents.html)

(Ware et al 1997) Colin Ware, Glenn Franck, Monica Parkhi and Tim Dudley. "Layout for Visualizing Large Software Structures in 3D," Visual 97 Second International Conference on Visual Information Systems, San Diego, November, 1997, pp. 215-225.

BIOGRAPHICAL SKETCH

Andrea Goethals received a Bachelor of Design from the College of Architecture at the University of Florida in 1989. She worked for an architect in Athens, Georgia for a few years while also freelancing as an artist, and then took time off to travel around Europe. Seeing the difference between the great old cities in Europe and those she remembered from the US caused her to return to the University of Florida, this time to get a Master of Arts in Urban and Regional Planning in the hopes of improving US cities. While in the planning program she specialized in Geographic Information Systems, which led her to realize her strong interest in working with computers, and thus led her to get a Master of Science in the Department of Computer and Information Science and Engineering.

She intends to combine her interest and education in both design and computer science, working for a research division of a private or public organization, in the areas of interface design, visualization and communication. Her goal is to help advance the visual interfaces of software.