Gadget: A Tool for Extracting the Dynamic Structure of Java Programs

Juan Gargiulo and Spiros Mancoridis
Department of Mathematics & Computer Science
Drexel University
Philadelphia, PA, USA
e-mail: {gjgargiu,smancori}@mcs.drexel.edu

Abstract

Source code analysis and inspection does not provide enough information to describe the structure of an object-oriented program completely because there are components and relations that only exist during its runtime.

This paper presents a tool, called Gadget, that helps software engineers extract the dynamic structure of object-oriented programs written in the Java programming language. The tool uses program profiling, filtering, and graph clustering techniques.

In this work we show how Gadget is used to analyze a standard graphical user interface library for Java, called Swing. This library has a complex structure, part of which we expose using data gathered by Gadget during the execution of a simple Java program that uses Swing.

1. Introduction

As the size of a software system increases, so does the complexity of its structure. The use of object-oriented (OO) development techniques and languages helps programmers manage this complexity by supporting data abstraction, encapsulation, polymorphism, and reuse. However, the OO approach makes understanding the structure of these systems more difficult because of features such as dynamic binding and polymorphism.

Understanding the dynamic structure of a system is helpful during software maintenance. The dynamic structure of an OO program shows which objects are created and what messages are sent between these objects at runtime. Dynamic analysis can be used to complement static source-level inspection and analysis, which may not provide all of the information software engineers need in order to understand an OO system. For example, the *Factory* OO design pattern [3] is used to "manufacture" objects and make these objects accessible to client objects through an abstract interface at runtime. Static analysis, using a tool such as Chava [6], would only reveal part of the complete design. Specifically, it would reveal the relationship between the client class and the Factory class but not between the client and the objects that were created by the Factory at runtime.

This paper describes a tool, called Gadget, to help software engineers extract the dynamic structure of Java programs. In the profiling phase the tool gathers runtime data from the executing program. This data captures class loading, object creation, and method invocation events. In the analysis phase the tool uses filtering and abstraction techniques to select classes of interest. The instances of these classes and their runtime relations are then displayed as a graph.

In general, one or more features of the target program is analyzed at a time (e.g., the *Save* feature of a word processing program). The recording of events that occur during the execution of a feature is called *tracing*, and the collection of trace data is called a *profile*. The profile is analyzed by the tool and the dynamic structure of the objects and relations involved in the trace is shown as a DDG (Dynamic Dependency Graph). A DDG is a graph where nodes represent classes or objects and edges represent relations (i.e., method invocation) between two objects or between a static class and an object. A method invocation can be an object creation (i.e., call to a constructor method) or a simple method call. Gadget uses a tool from AT&T Research called *dotty* [4] to display the DDGs. An example of a DDG is shown in Figure 7.

Additionally, Gadget uses a clustering algorithm to partition the DDGs, thus presenting the dynamic structures in a modular fashion that is easier to understand.

Figure 1 shows the architecture of Gadget, which consists of three main components: Control, Repository, and Profiler. The Control and Profiler components execute on different Java Virtual Machines (JVMs). The Control component contains Gadget's execution and analysis engines (i.e., Gadget Executor and Gadget Analyzer). The execution engine is in charge of the execution of Gadget and has some control of the execution of the target program being

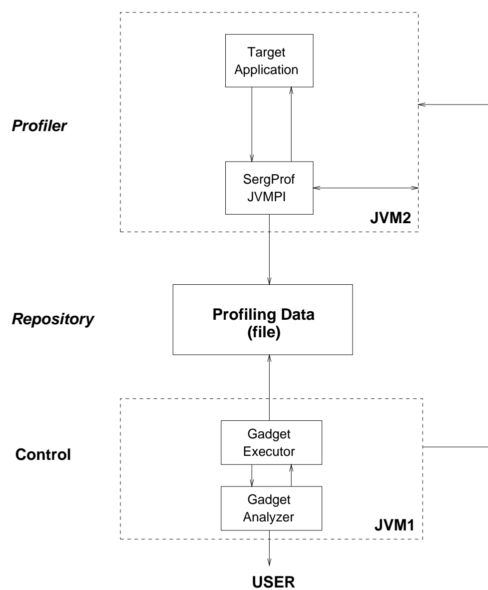

Figure 1. Software Architecture of Gadget.

analyzed. The analysis engine is in charge of managing the analysis process logic. This includes the interaction with the repository and the interaction with the user interface. Gadget's repository is an external file that contains data about the execution of the target program. This file is first written by the profiler and then read by the analysis engine. The Profiler component is in charge of running the target program through the *SergProf* profiling interface, which is our implementation of Sun Microsystem's Java Virtual Machine Profiling Interface (JVMPI) [13]. *SergProf* traces class loading, object allocation, method invocation and stack information at the time an object is allocated.

The structure of the rest of this paper is as follows: Section 2 provides an overview of related research. Section 3 provides background information about the Pluggable Look and Feel (PLAF) feature of the Java Swing library. Section 4 uses Gadget to extract the structure of the PLAF feature. This section presents the basic features of Gadget and shows screen captures of the tool, as well as the results of the PLAF feature's analysis. Section 5 concludes with a summary of the paper, an outline of the contributions made, and a list of the limitations of the current version of Gadget.

2. Background

In this section we outline the body of work that is most closely related to the work presented in this paper. At the end of the section we state the distinguishing characteristics of our work.

Jerding's [5] approach to supporting OO program understanding is based on visualizing object interactions and instantiation patterns during the execution of a program. His approach requires tracing the execution of the program being analyzed and then visualizing the dynamic data captured by the trace. This work is concerned with searching for solutions to the problem of visualizing large data sets of runtime data. It also focuses on detecting inconsistencies between the design and implementation of software systems.

De Paw, Helm, Kimelman and Vlissides [12] have created a library for extracting and animating the behavior of OO systems. They present summary information about the execution of a program using a chart-like visualization. They also present techniques to instrument OO programs so that their executions can be traced.

Richner, Duccasse and Wuyts [10] use logic programming to filter data that is generated by execution traces. Their approach also requires code instrumentation.

Shilling and Stasko [11], and later Stasko and Mukherjea [9], worked on animating designs and algorithms, which can be helpful during the design process. Unlike the other work presented in this section, their work does not involve program tracing and does not take a reverse engineering approach.

What distinguishes our work from the above research is the emphasis on filtering and clustering the trace data before producing design visualizations. Unlike Jerding's work, which tries to reduce complexity by providing better visualizations, our approach attempts to reduce complexity by filtering out only the runtime data that is relevant to the program feature being analyzed.

Automatic clustering was identified by Belady and Evangelisti [1] as a means to produce high-level views of the structure of software systems. We use the *Bunch* [8] clustering system to group sets of objects and their interrelations in order to produce a high-level "road map" of the dynamic software structure. Another distinguishing characteristic is that our work does not rely on program instrumentation but rather on capturing trace data directly from the Java virtual machine using the JVMPI [13]. Finally, we are treating this work as an ongoing project and hope to receive feedback from users. For this reason we provide a free copy of Gadget from our web page (`http://serg.mcs.drexel.edu/gadget`) and intend to maintain the tool if a significant user community materializes.

Before we show how Gadget is used to extract the structure of Swing's PLAF feature in Section 4, we briefly provide some background material on this feature in the next section.

3. The Pluggable Look and Feel Feature of the Swing Library

Swing is a Java library to support the development of graphical user interfaces for Java programs. One of Swing's goals is to support multiple platforms while preserving a consistent look and feel.

The Swing pluggable look-and-feel (PLAF) design is the portion of a GUI component's implementation that deals with the presentation (the look) and event-handling (the feel). Note that the LAF can be changed at runtime [2]. Figure 2 shows three different GUI buttons, each with a different LAF.

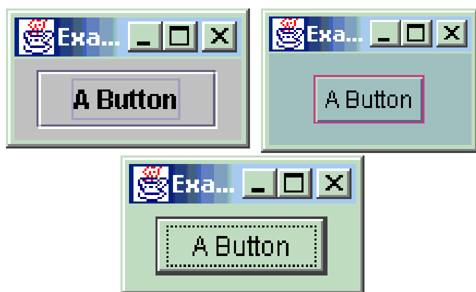

Figure 2. Example program with three different LAFs (default, Motif and Windows).

Swing enables programmers to change the LAF of a GUI dynamically by means of a simple interface as shown in the following source code fragment, where the LAF is changed by using a command line argument.

```
...
5 if (args.length > 0 && args[0].equals("motif"))
6   UIManager.setLookAndFeel(
     "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
7 else if (args.length > 0 && args[0].equals("windows"))
8   UIManager.setLookAndFeel(
     "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
...
```

As shown in lines 6 and 8 of the source code, the `UIManager` class is used change the PLAF dynamically. In order to explain the `UIManager` class let us consider the process for creating a `JButton` component (see Figure 3).

The creation of the `JButton` component includes the installation of a corresponding UI delegate through the `UIManager` class. The `UIManager` then gets the name for the UI delegate class for this component, depending on the installed LAF. For example the UI delegate class of the Windows LAF `JButton` is `WindowsButtonUI`. Finally, a UI delegate object is created and the `JButton` component obtains a reference to it. This process is called *UI Installation*.

The UI delegate is installed by the `UIManager` class, which is a static class that has knowledge of the current LAF. The `UIManager` class makes use of the `UIDefaults` class, which has methods for accessing specific types of information about look-and-feel. The `UIDefaults` class includes, among other things, default values for presentation-related properties (such as color, font, border, and icon) for each UI delegate.

Figure 3 shows the class diagram for a program that uses a `JButton` with a PLAF. This diagram describes Swing's structure (for one component) showing the relations between the different classes. Examples of inheritance relations are the large edge relations between `MotifButtonUI` and `WindowsButtonUI` with `BasicButtonUI` (they both extend `BasicButtonUI`). The dotted edge relation between `JButton` and `UIManager` is an example of a relation between classes that depend on static calls made in the process of installing a PLAF to a component. The dashed edge relation between the `JButton` class and each of the specific LAF classes (`MotifButtonUI` and `WindowsButtonUI`) is an example of a runtime relation.

4. Analysis of Swing's PLAF Feature Using Gadget

This section shows how Gadget can help programmers extract the structure of the PLAF feature of Swing, which was described in the previous section.

As an example we will analyze a small program that displays a window with a button. The program contains only one class, called *PLAFTester*, which is also the name of the program. The *PLAFTester* program loads a `JFrame` and adds a `JButton`. To show a correlation between the extracted structure of a `JButton`, as produced by Gadget, and the design described in Figure 3, the Windows look and feel was set during runtime.

We next go through the step-by-step process of extracting the structure of the PLAF feature using Gadget. Throughout the explanation of this process, screen shots of the tool will be shown. The Gadget GUI has two main panels (see Figures 4-6, 8). Each panel is selected by clicking on a tab. One tab reveals the tracing panel and the other the analysis panel. The analysis panel uses a "wizard" style interface, which helps users go through the correct sequence of steps via **Next** and **Back** buttons.

4.1. Step 1: Tracing

The first step in the analysis is to run *PLAFTester* using Gadget's tracing capabilities. The execution involves starting the program until it displays a window. No further trac-

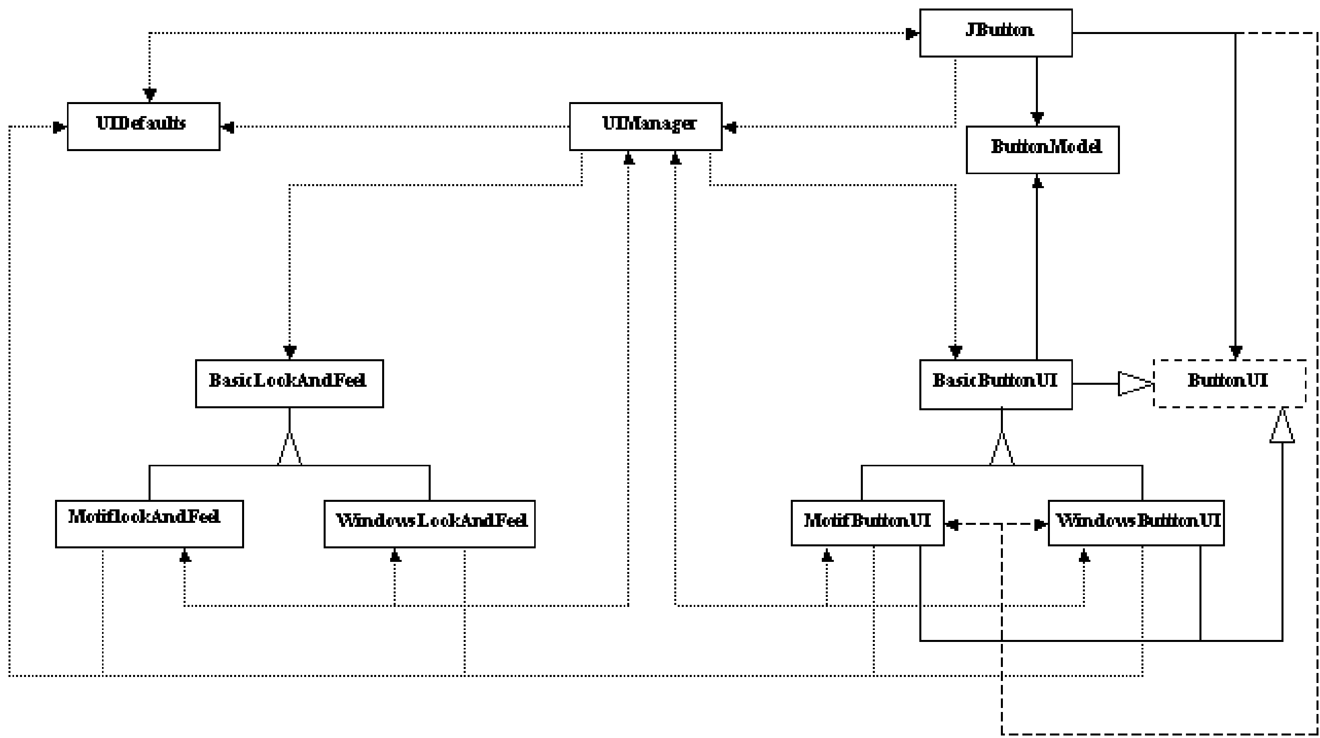

Figure 3. Architecture of the `JButton` UI Installation Process

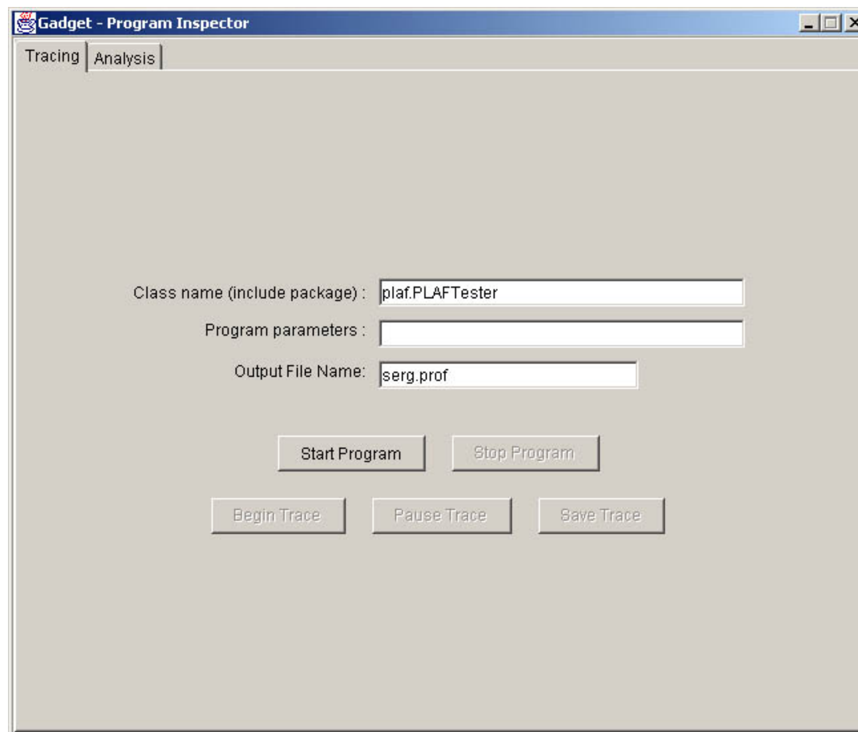

Figure 4. Profiling Screen

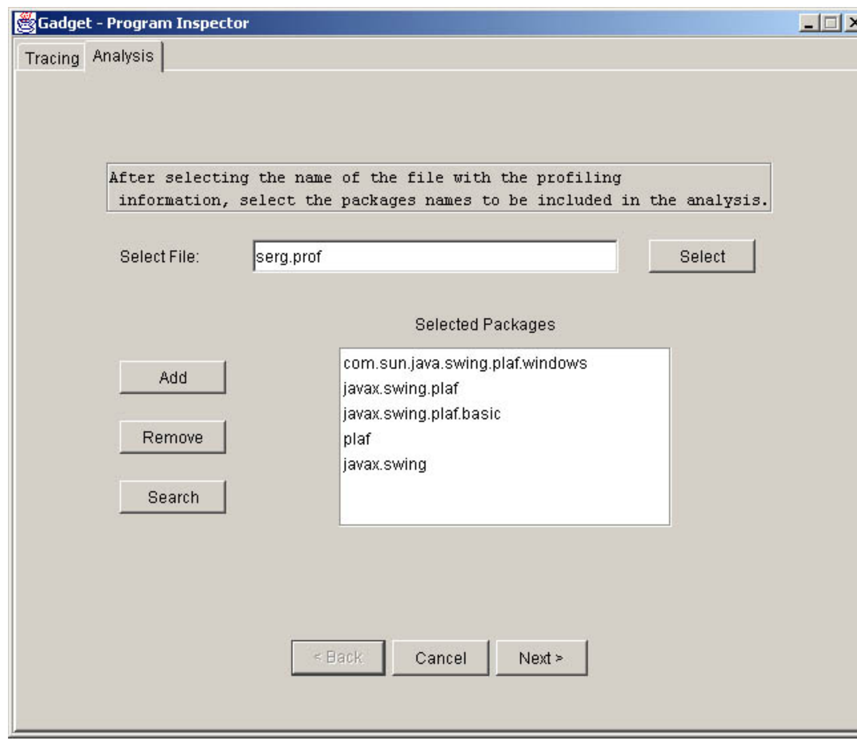

Figure 5. Package Filtering Screen

ing is needed because all of the objects related to the PLAF are already loaded when the window is displayed.

Figure 4 shows the name of the class containing the main method in the text field. The user can start running the target program by pressing Gadget's **Start Program** button. This action starts executing the program by displaying a new window (if the program has a GUI) or sends the output to the console from where Gadget was started.

After the program starts running, the user can begin tracing by pressing the **Begin Trace** button. By doing this, the profiling interface starts tracing and the trace data is dumped to the profile file (e.g., *serg.prof*). By pressing the **Pause Trace** button, the profiler stops generating data. This option is useful when the user wants to analyze program features one at a time.

4.2. Step 2: Selecting Packages

Once the trace is performed, Gadget's analysis tool is started. The first step in the analysis is to select the packages to be included in the analysis.

Figure 5 shows a screenshot of the first analysis screen. On this screen the user specifies the profile file name and then the package names to include in the analysis. Users may specify which packages are to be included in the analysis either manually, using the **Add** and **Remove** buttons, or automatically using the **Search** button. The list in the center of the screen is the **packages list**, which shows the selected packages containing the classes to be analyzed.

4.3. Step 3: Selecting Classes

After selecting the packages, the next step is to select the classes from the selected packages. By pressing the **Next** button on the **Package Filtering Screen**, Gadget performs the filtering of the profile. After the filtering is complete, the **Class Filtering Screen** is displayed, showing the classes that participated in the execution.

The screen presented in Figure 6 contains two lists. One contains the selected classes to be included in the analysis and the other contains the classes that will be ignored.

4.4. Step 4: Showing the Dynamic Dependency Graph

After selecting the classes, Gadget displays a DDG, which represents the runtime relations as edges and the objects as nodes. The graph is shown in Figure 7; dark nodes represent classes that were not instantiated, but that interact with objects via static methods, and light nodes represent objects that were created at runtime.

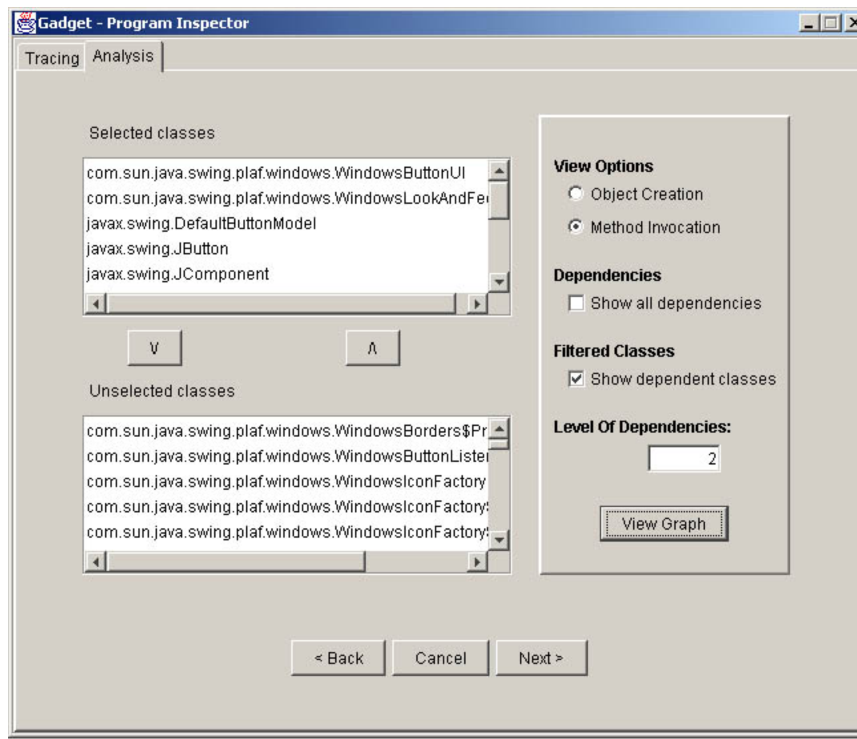

Figure 6. Class Filtering Screen

Figure 7 essentially presents the same information as Figure 3 from a runtime perspective. The numbers on the edges of the DDG in Figure 7 indicate the order of the method invocations between the objects at runtime. For example, the `JButton` first sends a message to the `UIManager`, then the `UIManager` sends a message to the `UIDefaults` object, and so on. Unlike Figure 3, Figure 7 does not show inheritance relations because they are static not dynamic relations.

In most cases, software engineers do not have access to documentation about the OO design of the system they are trying to understand. In the absence of such documentation, a tool such as Gadget can provide one of more DDGs that can serve as a good starting point for the program understanding process.

4.5. Step 5: Performing Clustering on the DDG

By pressing the **Next** button in the **Class Filtering Screen**, the **Clustering Screen** (Figure 8) appears. Before showing this screen, Gadget calls the *Bunch* [7] tool to cluster the current DDG. The *Bunch* algorithm partitions the DDG so objects that are tightly linked via relations are grouped together, as such objects are more likely to be closely related. The algorithm also tries to minimize the number of relations across cluster boundaries. In essence, the algorithm balances the tradeoff between tight intra-cluster cohesion and loose inter-cluster coupling.

In Figure 8 the user can see the clustered DDG by pressing the **View Graph** button. This view shows all of the clusters and the objects and relations in each cluster. If the **View Cluster** button is pressed, the classes within the selected cluster are displayed in the graph using *dotty*. This filtering option is useful when working with a large number of objects, whose complex DDG may be difficult to read. Gadget also shows the clusters using a directory tree representation, where each folder in the tree represents a cluster (see Figure 8). Users may show or hide the contents of each cluster by pressing the plus or minus sign on each folder icon, respectively.

5. Conclusions

We described a tool called Gadget and showed how this tool can be used to extract the runtime structure of OO programs written Java. In order to study the PLAF feature, we applied Gadget to a small program that uses a Swing `JButton`.

The paper concludes with the technical and research contributions of this work along with a description of this work's limitations, which present opportunities for future work.

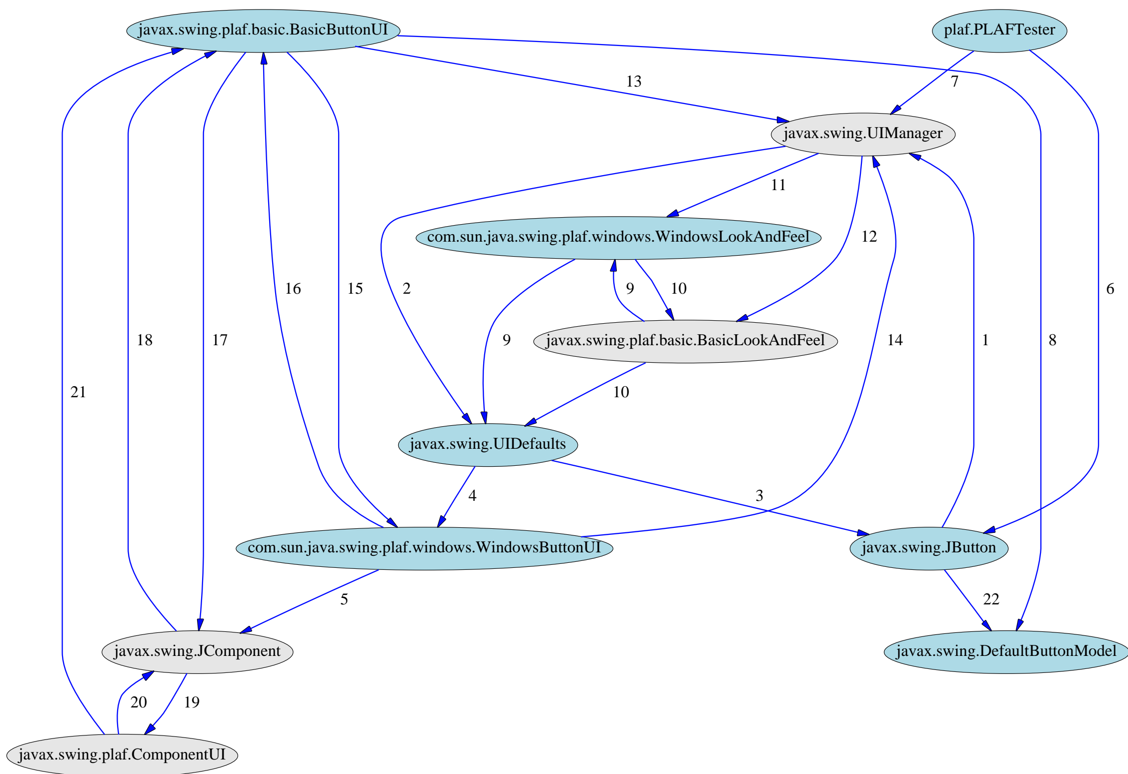

Figure 7. The DDG of the *PLAFTester* program.

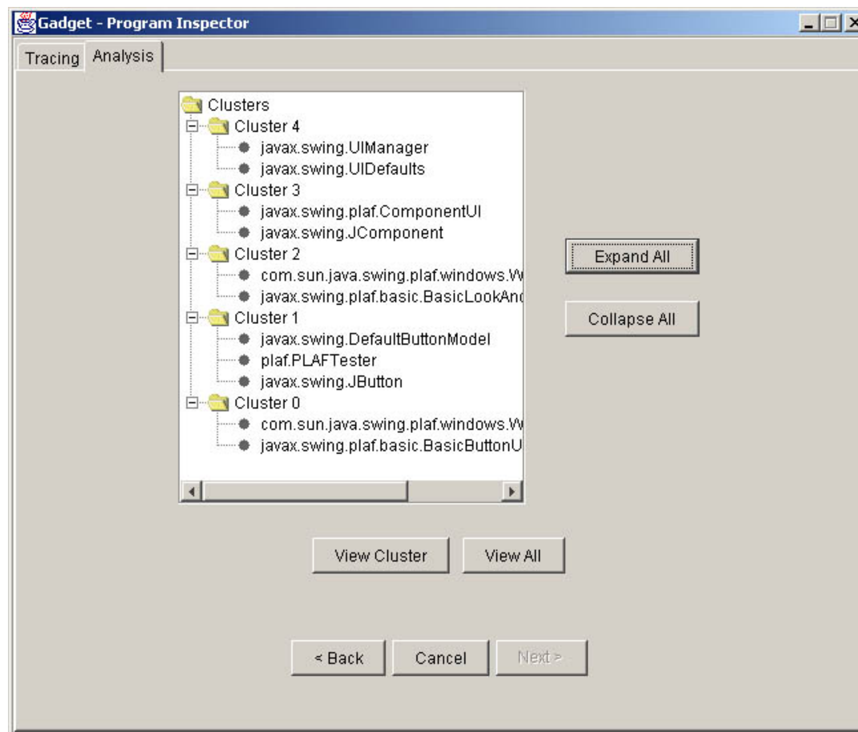

Figure 8. Clustering Screen

5.1. Contributions

- Gadget's filtering features for reducing the size of program execution traces are essential when dealing with complex OO software. For the most part, the tool's filtering techniques take advantage of the way Java programs are organized into packages and classes. In addition to package and class filtering, the tool supports another type of filtering that enables users to trace individual features of a program by using Gadget's **Begin Trace**, **Pause Trace**, and **End Trace** buttons.
- Gadget uses software clustering techniques to make large dynamic structures more modular and, thus, easier to understand. Most of the work on software clustering has been on clustering static code artifacts such as methods, classes, modules, and files.
- Gadget allows software engineers to analyze Java programs without instrumenting the source code.
- Gadget is available, free of charge, from our web page.

5.2. Limitations

- Gadget uses files to store traces, thus, its performance is poor when large profiles are analyzed.
- The current version of Gadget only runs on the Windows platform because *SergProf* is implemented as a Windows dynamically linked library.
- Gadget only works with Java programs. One of goals is to support C++ in the near future.

Acknowledgments

This research is sponsored by two grants from the National Science Foundation (NSF): a CAREER Award under grant CCR-9733569 and an Instrumentation Award under grant CISE-9986105. Additional support was provided by grants from the research laboratories of AT&T and Sun Microsystems. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, the U.S. government, AT&T, or Sun Microsystems.

References

[1] L. Belady and C. Evangelisti. System partitioning and its measure. *Journal of Systems and Software, 2*, pages 23–29, 1981.

[2] A. Fowler. A swing architecture overview: The inside story on jfc component design. *http://java.sun.com/products/jfc/tsc/articles/-architecture/index.html*, August 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison -Wesley*, 1995.

[4] E. Gansner, E. Koutsofios, S. North, and K. Vo. A technique for drawing directed graphs. *Transactions on Software Engineering*, 19(3):214–230, March 1993.

[5] D. F. Jerding. *Visualizing Interaction Patterns In Program Executions*. PhD thesis, Georgia Institute of Technology, November 1997.

[6] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. *Working Conference on Reverse Engineering*, 1999.

[7] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. *IEEE Proceedings of the 1999 International Conference on Software Maintenance (ICSM 99). IEEE*, 1999.

[8] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. *In Proceedings of the 6th Intl. Workshop on Program Comprehension*, June 1998.

[9] S. Mukherjea and J. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction, Vol. 1 No. 3*, pages 215–244, 1994.

[10] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. *In Proceedings of the International Conference on Software Maintenance (ICSM), Oxford*, pages 13–22, September 1999.

[11] J. Shilling and J. Stasko. Using animation to design, document and trace object-oriented systems. Technical Report GIT-GVU-92-12, Graphics, Visualization, and Usability Center, Georgia Institute of technology, College of Computing, Atlanta, GA, June 1992.

[12] R. H. W. De Paw, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. *In Proceedings of the ACM OOPSLA '93 Conference, Washington, D.C.*, pages 326–337, October 1993.

[13] www.javasoft.com. Jvmpi - java virtual machine profiling interface. *http://java.sun.com/j2se/1.3/docs/guide/jvmpi/jvmpi.html*, 2000.