A Design Recovery View - JFace vs. SWT

Technical Report 2009-564

Manar Alalfi
School of computing- Queen's University
Kingston, Ontario, Canada
alalfi@cs.queensu.ca

Abstract

This paper presents an experience report on using software engineering and design recovery tools to gain understanding and insight into the design of a software system without being burdened by a large source code repository. It is often the case that software systems contain incomplete or inaccurate documentation, or that the software designers did not follow software engineering best practices entirely during the development process. These tools extract and visually present design information from these software systems, so that one can understand the interconnections of the modules that comprise that system.

1. Introduction and Motivation

Large volumes of code are typically difficult to understand. This problem is made worse because most software systems are not documented. Studies show that about half of the time spent making changes to software is spent on understanding the software. Automated reverse engineering techniques help developers by minimizes the amount of manual source code analysis needed to understand a system. Reverse engineering is the process of analyzing a subject system to identify its components and their relationships, creating a representation of the system at a higher level of abstraction than what exists in the source code. Specifically, reverse engineering provides an aid to the comprehension of complex systems.

In this project an attempt to understand two large system was made using a number of design recovery tools, then a kind of analysis was made in order to compare some aspects of the two systems, the systems that are chosen : The Standard Widget Toolkit (SWT) and JFace libraries which are used to develop graphical user interfaces (GUIs) for the Eclipse environment.

2. Systems Overview

- **SWT**

  The Standard Widget Toolkit (SWT) was released in 2001, integrated with the Eclipse Integrated Development Environment (IDE). It has since become an independent release, available for download by itself. The Eclipse.org consortium thought both AWT and Swing to be inadequate for building real-world commercial applications, due to the lowest common denominator approach of AWT and the sluggishness and look-and-feel problems of Swing. SWT takes the "best of both worlds" approach between AWT and Swing: native widgets where available, and java implementations where unavailable.

Per	Packages	Classes	Functions	Lines of Code	Javadocs	Lines of Javad
Project	18	484	6,073	69,868	2,817	37,845
Package		26.89	337.39	3,881.56	156.5	2,102.5
Class			12.55	144.36	5.82	78.19
Function				11.5	0.46	6.23

figure 1.1 SWT Statistics

- **JFace**

  JFace is a UI toolkit built on top of SWT. It uses SWT widgets to implement GUI code that is common among different applications. Using JFace, you can create user interface components with less code than if you had started at the basic SWT widget level. For example, to create a message dialog, you can either use the SWT widgets directly or you can use the classes provided by JFace. Using SWT, you would create the shell for the dialog, set its size and location, add and position buttons and labels, and maybe even add an image. With JFace, you could make the same dialog by calling one method with three parameters, the rest of the details are handled by the JFace dialog class. This requires less effort and less code each time you use it in an application.

Per	Packages	Classes	Functions	Lines of Code	Javadocs	Lines of Javad
Project	15	263	3,162	21,306	2,767	24,318
Package		17.53	210.8	1,420.4	184.47	1,621.2
Class			12.02	81.01	10.52	92.46

figure 1.2 JFace statistics

- **JFace and SWT**

The lines between SWT and JFace are much cleaner. SWT does not depend on any JFace or platform code at all. Many of the SWT examples show how you can build a standalone application.

JFace is designed to provide common application UI function on top of the SWT library. JFace does not try to "hide" SWT or replace its function. It provides classes and interfaces that handle many of the common tasks associated with programming a dynamic UI using SWT.

The relationship between JFace and SWT is most clearly demonstrated by looking at viewers and their relationship to SWT widgets.

3. Report scope:

Because the systems that are chosen were considered to be large systems, SWT compose of 444 file and 69,868 line of code, and JFace is composed 250 file and 21,306 line of code , so describing all the differences between the two systems exceeds the project size limit, so what I focus on is to understand that JFace is really an extension of SWT, and what are the libraries that JFace are using from SWT, then to focus on tow of the improvements that JFace added on SWT, with respect to simplification of GUI coding for Dialog boxes and viewers, the main objective from this project is to work on the process of understanding large systems, using different design recovery tools, and to do some analysis on the output that is gained from those tools, to see how much those tools facilitate the process of understanding large systems in order to simplify the job of future maintenance of them.

1. Project Phases and Tools description

During the process of understanding the systems a number of design recovery tools were used, the process that is used is depicted in the following figure:

A detailed description of the tools that are being used is as follows:

- **Code Analysis**

The current trend is to build repositories from a system's source code so that those repositories can be used for a variety of reverse engineering analysis. The repositories are useful because complex reverse engineering tools can be built by analyzing information stored in the repository without parsing the system's source code.

Two popular approaches exist to construct software repositories. One is to store variants of abstract syntax trees in the repository; the other is to structure the repository as a relational database. Once the repositories are constructed from the source code, queries can be made to the repositories in order to exact structural information about the source code.

Over the past several years, the AT&T Research Lab has developed a family of source code analysis tools such as Acacia for C and C++ and Chava for Java.

In these tools, software systems are represented as collections of entities that refer to each other. An entity represents a static syntactic construct such as a macro, a type, or a function. The output of Acacia and Chava are two repositories, i.e., the entity repository and the relationship repository. The entity repository stores the entities such as file names, variables, functions, classes along with attributes, such as scopes, line positions, and so on. Likewise the relationship repository stores the relationships between entities, such as function calls, inheritance, and variable references. A variety of Unix command line tools are available to query against the repository, answering questions such as:

- Is variable a defined in file b?

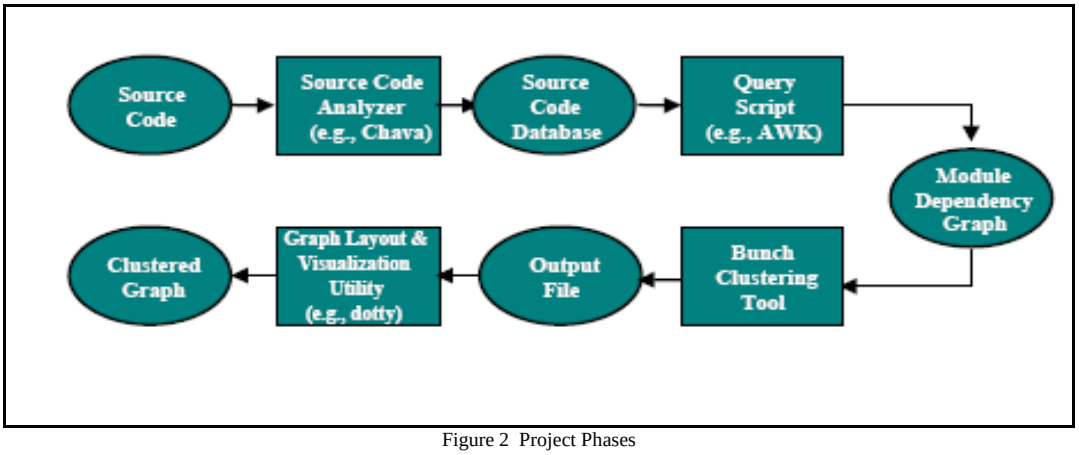

Figure 2 Project Phases

- Is function a referenced by function b?
- What are the child classes of class a?

Advanced analysis such as dead code detection and reachability analysis can be applied to the repositories as well. The query results are display in text format.

The work described in this project uses Chava to create the repositories.

- **Design Extraction**

Software systems often need to be modified to improve performance, add new features, adapt to new platforms or hardware, and so on. To modify a software system, developers have to understand the system. As the size and complexity of software systems increase, the time spending on understanding software systems increases as well. In most cases the relevant design documentation is missing or inconsistent, making the problem even worse. Therefore, tools that can provide a high-level system decomposition become very helpful to facilitate the comprehension of software systems.

The principle artifact that must be examined is the system's source code. Thus, the major task in software reverse engineering is to build an abstract model of a software system from its source code.

Mitchell and Mancoridis developed a software tool called Bunch , which automatically decomposes the structure of software systems into subsystems. Modules with high cohesion are grouped in the same subsystems (clusters), and independent modules are grouped into separate subsystems. The modules and dependencies of a system are mapped to a Module Dependency Graph (MDG) using source code analysis tools such as Acacia for C and C++ and Chava for Java. The goal of Bunch is to find a good partition of an MDG graph. It is the first system to employ generic search algorithms to the software clustering problems. Mitchell and Mancoridis introduced an objective function called Modularization Quality (MQ). The MQ rewards the creation of highly cohesive clusters, and penalizes excessive coupling between clusters.

Hence, Bunch reformulates the software clustering activity into an optimization problem where the goal is to maximize the value of MQ. The assumption behind this rationale is that most software systems are designed in such a way that highly cohesive modules are organized into the same subsystem while loosely coupled modules are organized into separate subsystems.

The process is conducted automatically. Also, users can integrate their knowledge with clustering process by assigning some modules to subsystems manually. Extensive case studies and experiments show that Bunch does a good job of producing a subsystem decomposition with or without knowledge of the software design. Bunch also

includes a programmer's Application Programming Interface (API) so that the clustering tool can be integrated with other tools.

- **Graph Drawing for Software Visualization**

Visual presentations can ease the understanding of complex systems. Not surprisingly, extensive research has been conducted on how to store, layout and display graphs.

Barghouti and Mocenigo developed an extensible graph drawing package written in Java, called Grappa. It consists of a set of classes that implement graphs, in addition to representation and presentation services. It also provides an API so that it can be integrated into applications that require graph drawing, editing, and browsing. The second version of Grappa, in addition to supporting the feature of bird's eye view, is able to handle large graphs, which the first version of Grappa could not. Grappa invokes the dot, a graph layout tool. Dot, which runs fast enough for interactive use, uses a four-pass algorithm for drawing directed graphs. Dot is a command-line utility that takes a dot description file as input, and produces an output file where the nodes are assigned a position in a 2D space based on layout properties.

By default, dot positions nodes to minimize edge lengths and edge crossing. Grappa renders a graph in a Java applet based on the layout information produced by dot.

dot can also transform a dot description file into a formatted graph using a number of standard image file formats such as GIF, PS, JPEG, and PDF. The dot description file is a text file where users specify the edges and nodes to appear in the output graph. Users are able to control the font type, font size, colors of nodes and edges, shapes of nodes, labels and so on. User may also provide information that dot uses in the layout process.

2. Steps and experiment parameter that are being used in order to use the different tools to gain understanding of the analyzed systems

A. To create the *.A files which are intermediate files used by Chava to perform its analysis the following command was used:

chava –c *.class

B. To create the entity.db and relationship.db files which are the final output of the Chava tool the following command was used:

chava –l *.A

C. To create the mdg file the following command was used:

mdg –java –A –A > mdg_output.mdg

D. Now I start Bunch.

E. To select the MDG graph dependency file, I clicked on the **Select…** button next to the Input Graph field. I select ***mdg_output.mdg*** file that was just generated.

F. To select the name of the output file which will be created after the clustering is completed (i.e., the clustered graph), I clicked on the **Select…** button next to the Output Cluster File field. The extension of file will be appended automatically based on the type of output file format you will select to use. For example, all dotty files will be given the ".dot" extension, and all text files will be given the ".txt" extension.

G. To select the output file format, I chose from the dropdown list of Output File Formats. (i.e., text, dotty, Tom Sawyer) For this run, select ***Dotty*** as the output file format.

H. To select the Clustering Method, I select ***Hill Climbing*** as our clustering method.

I. For this run, I select ***Agglomerative Clustering.***

J. For this run, I left the ***Clustering Algorithm*** as what Bunch thinks is appropriate for the given MDG file.

K. I Press the ***Find*** button at the bottom of the pane. You should see the ***Library*** modules relocate to the right side list.

L. Once all of your options are selected, press the ***Run*** button to initiate the clustering process.

M. Once Bunch is done the clustering process, an output ***<filename>.dot*** file should exist in the earlier specified location.

N. I Typed in the following to view the ***<filename>.dot*** file using the dotty graphing tool.

dotty <filename>.dot

The graph generated in this step is too large to be insert into the document, so other tools , had been used to deal with large graph, and do the required zooming.

3. Experiments Results

After following the above steps in analyzing the two systems, I found that using Bunch clustering tool simplify the JFace system by clustering the relation between it's classes into 13 cluster. A separate cluster was assigned to the libraries which helps to remove the clutter of relations in the graph, Figure 6 shows the resulted 13 clusters, and figure 7 shows the set of libraries that are used by the system, where figure 10 shows the 15 cluster of the SWT system, and figure 11 shows the set of libraries that are used by the SWT system, by comparing the set of libraries of the two systems ,I recognized that all the JFace library files are either SWT files or used by SWT as a library files, the name of those files can be clearly identified from figure 16 which list all the SWT listeners, focusing our attention on the improvement that JFace added on SWT on coding Dialogs, we can recognize the difference from figure 8 for JFace and figure 12 for SWT , For example, to create a message dialog, using the SWT widgets , you would create the shell for the dialog, set its size and location, add and position buttons and labels, and maybe even add an image. With JFace, you could make the same dialog by calling one method with three parameters, the rest of the details are handled by the JFace dialog class. This requires less effort and less code each time you use it in an application, also the basic dialogs that are supported by SWT are:

colorDialog,FontDialog,FileDialog, printDialog, DirectoryDialog,messageDialog.

Where there are 6 dialog types directly inherited from the dialog class , and three others inherited indirectly, where the dialog class is inherited from window class , which in it's tern inherited from IShellProvider class, which is related to one of the libraries ShellAdapter, one of SWT adapter.

A window is the JFace class for a top level window -- in other words, one that is managed by the OS window manager. A JFace window is not actually the GUI object for a top level window (SWT already provides one, called a *Shell*). Instead, a JFace window is a helper object that knows about a corresponding SWT Shell object and provides code to help create/edit it, listen to its events, etc. Figure 3 shows the relationship between your code, JFace, and SWT.

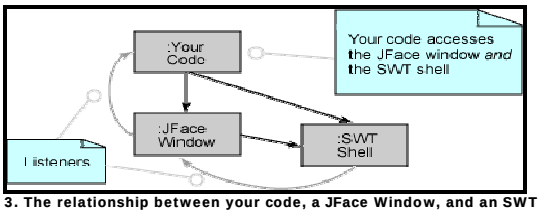

Figure 3. The relationship between your code, a JFace Window, and an SWT Shell

In fact, this model is the key to understanding how JFace works. It is not really a layer on top of SWT, and it doesn't try to hide SWT from you. Instead, JFace recognizes that there are several common patterns of use for SWT, and it provides utility code to help you program these patterns more easily.

To do this, JFace either provides an object that you use or a class that you can subclass (and sometimes it provides both).

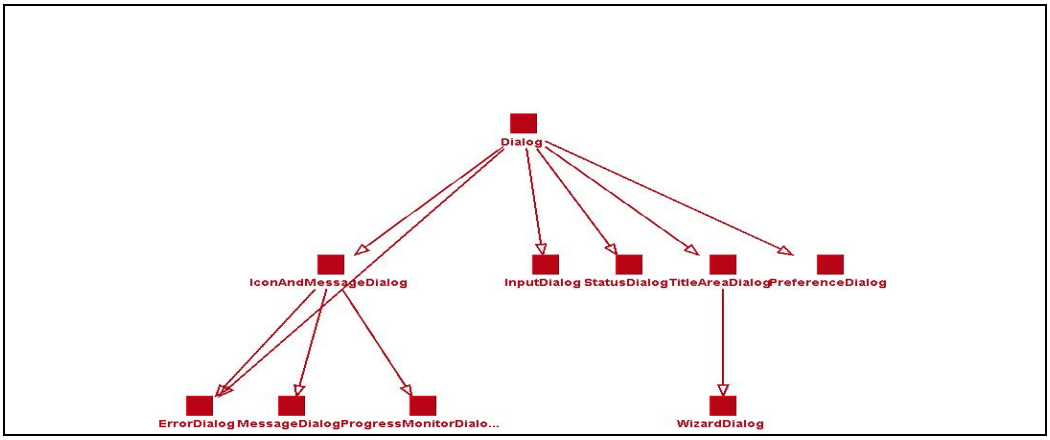

Figure 4 JFace Dialogs

- What if a change is being required to be done on the Messagebox class of the SWT library? , is this change will affect other classes? , and if so what are those classes that would be affected?
- What if a change is being required to be done on the messageDialog of the JFace library? Is this change will affect other classes? , and if so what are those classes that would be affected?

In order to be able to answer those questions, some kind of structured analysis is being done using an analysis tool produced by IBM (Structural analysis for Java).This tool can tell about the dependencies of classes on each other and represent that visually.

The result of applying the above two queries on SWT and JFace, shows that there will be no effect if we did a change on the MessageBox class for the SWT, this will not affect any class in the system, so it's safe to do such change, while if we intended to do a modification on the MessageDialog of the JFace library, 46 component will be affected, and this reflect that the improvement added on JFace library in order to ease the coding of GUI, increase the complexity of the system, it increase the degree

of coupling between the system classes, the result of this analysis could be seen from the following figures:

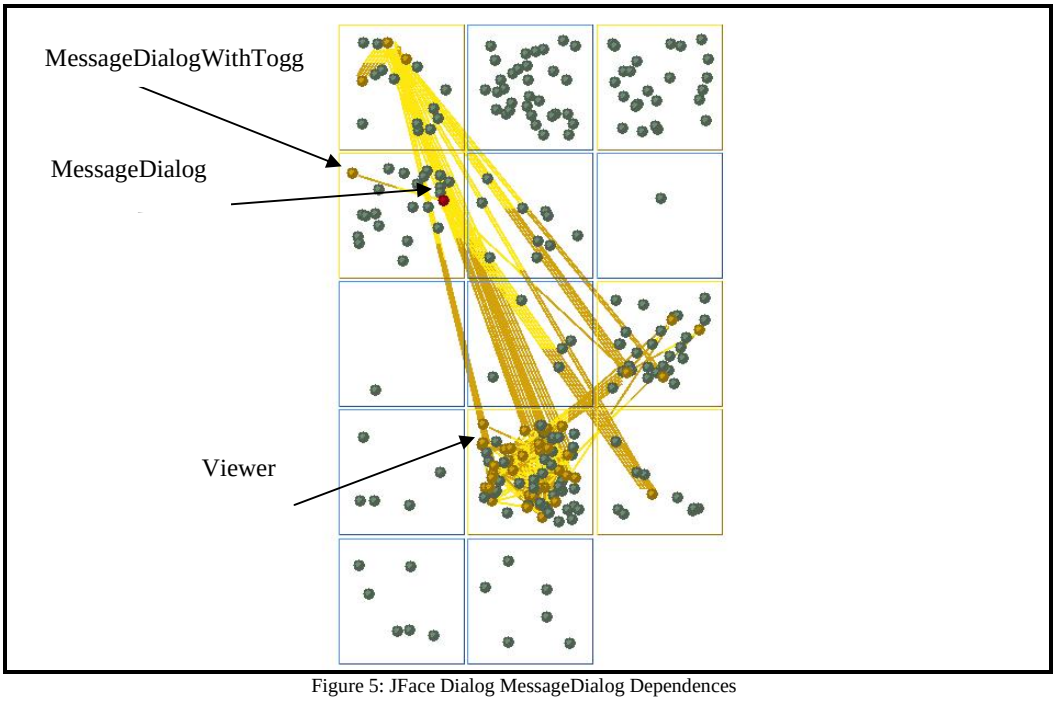

Figure 5: JFace Dialog MessageDialog Dependences

We can see from the above figure that there is a high dependency between the MessageDialog class and the set of Viewer classes, this can direct us to the most part of the system that is being affected with the change, the tool also gives the name of the 46 component that are affected, where there is no place to include them in this report, but we can recognize from the figure that one of those classes are from the same group of Dialog class, this one which is directly inherit from the class MessageDialog, specificallyMessageDialogWithToggle.

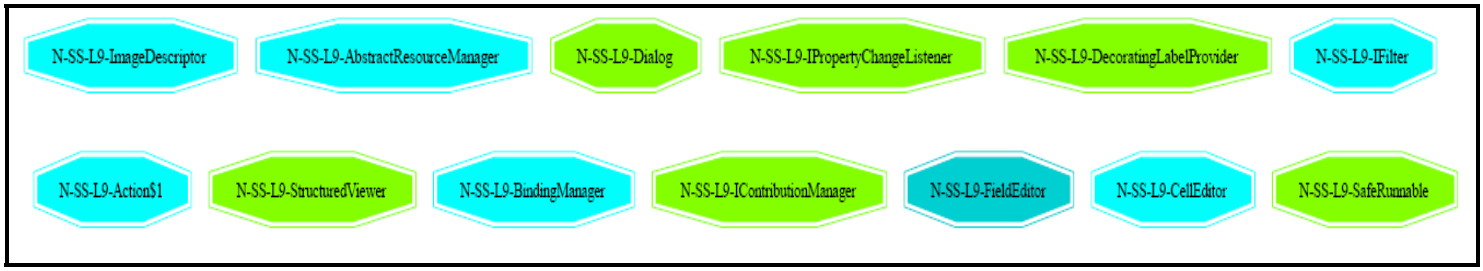

Figure6: the output for the clustering process of JFace System

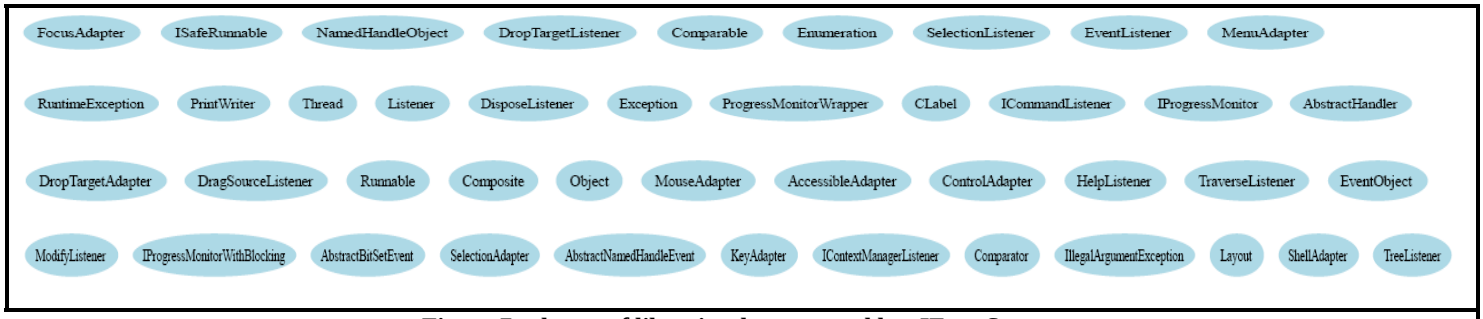

Figure 7 : the set of libraries that are used by JFace System

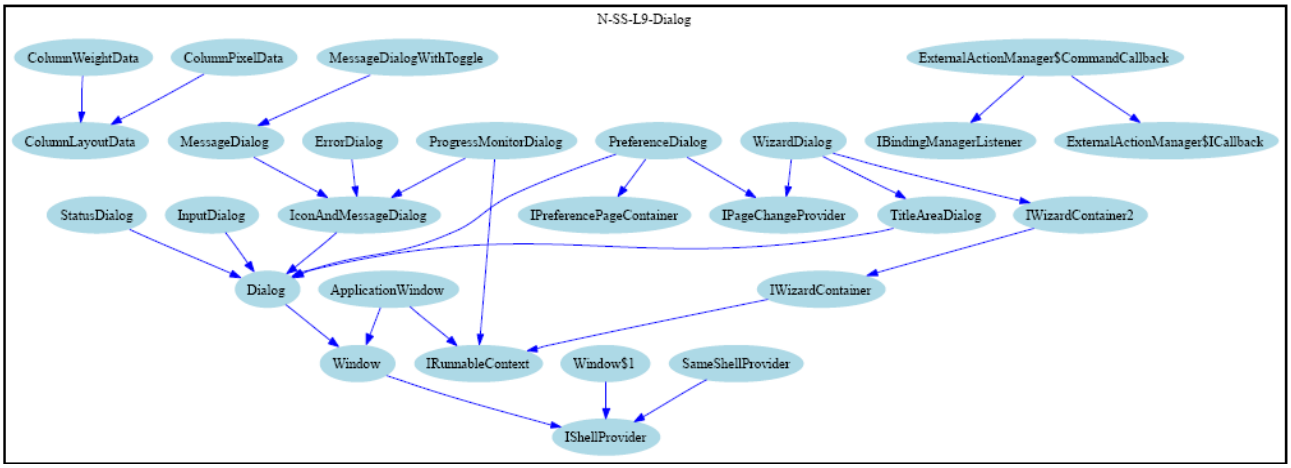

Figure 8: the Dialog cluster of JFace System

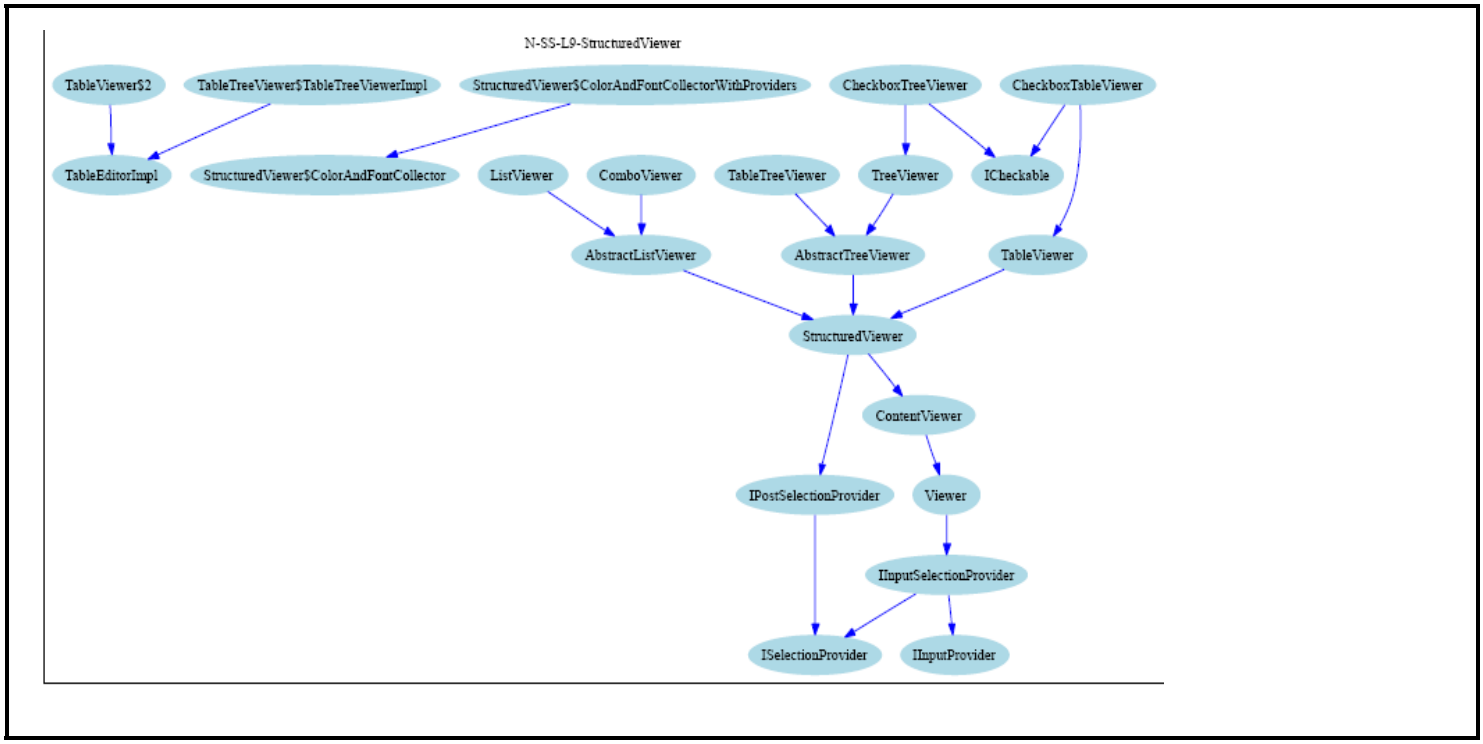

Figure 9 : the structuredViewer Cluster for JFace System

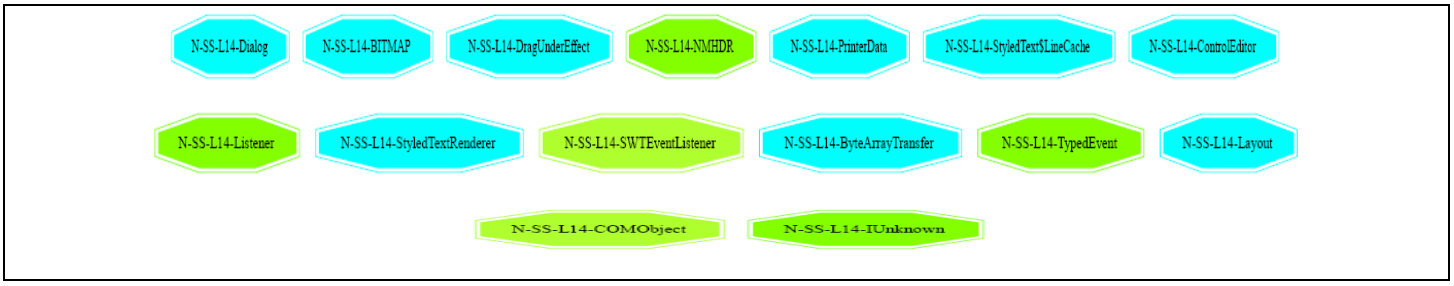

Figure 10: the output clusters for the SWT System

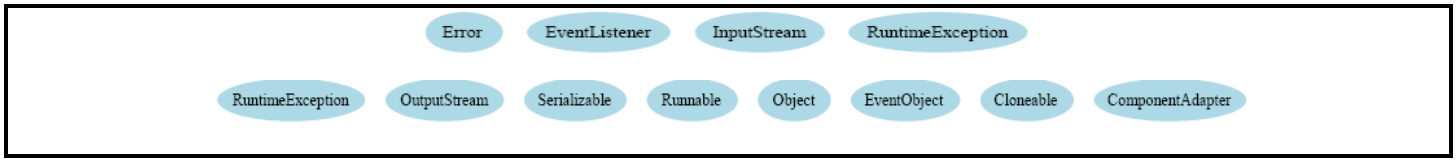

Figure 11: the library files that are used by the SWT System

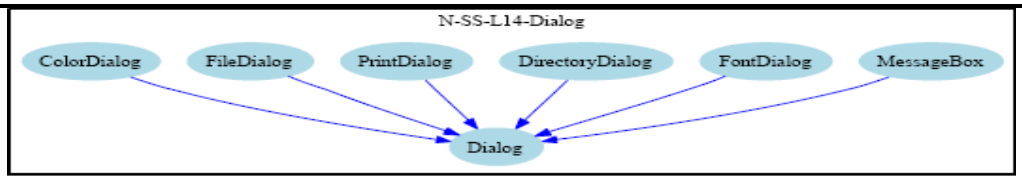

Figure 12: the Dialog cluster for SWT system

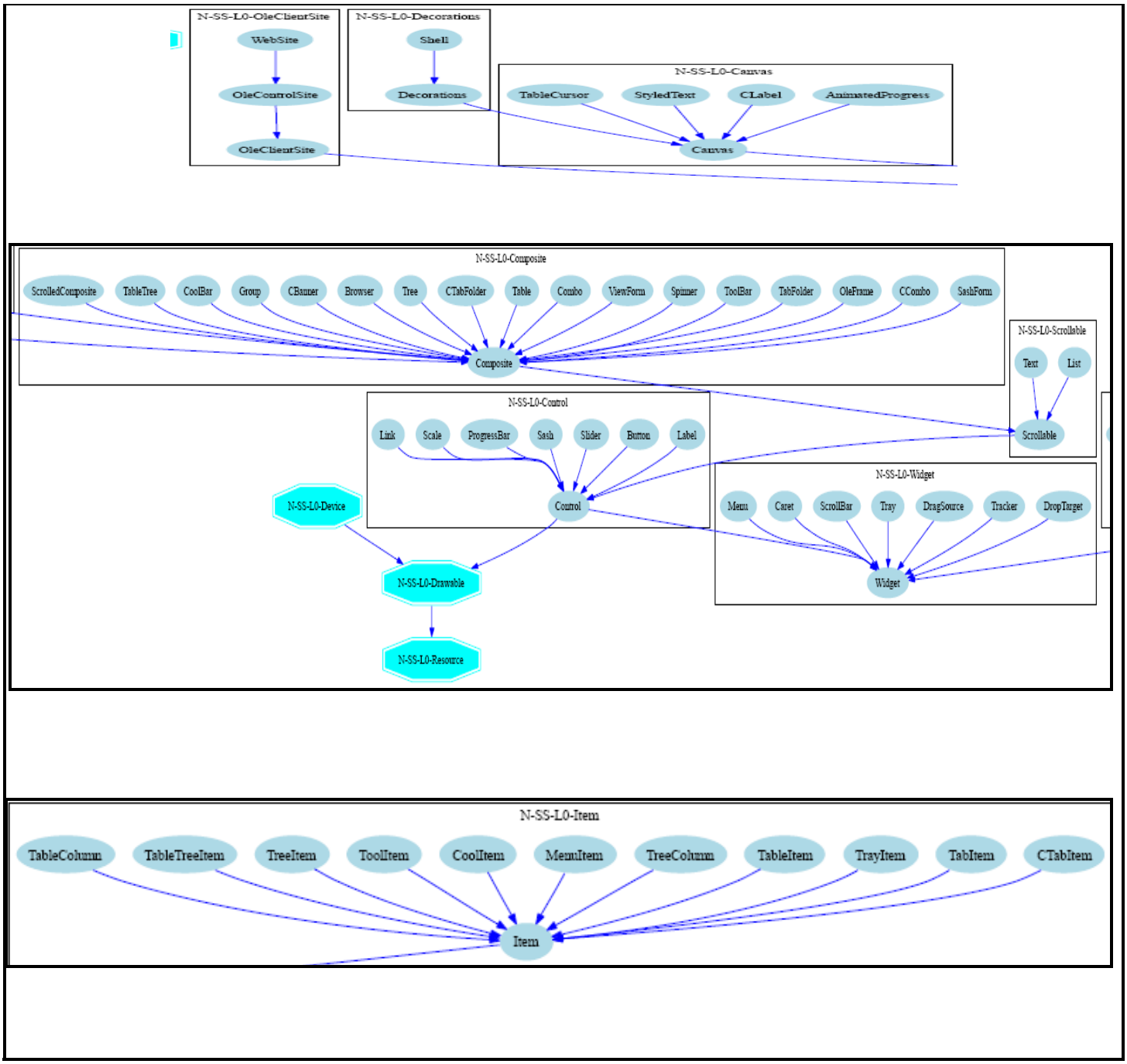

Figure 13 : the COMObjec Cluster for the SWT System (notice the widget Class)

JFace Viewers

Viewers display an object using different SWT widgets. The concrete viewer types available for quick use are CheckboxTableViewer, CheckboxTreeViewer, ListViewer, TableTreeViewer, TableViewer, and TreeViewer. These concrete viewers have built-in support for filtering and sorting, those types of viewers are viewed in figure (9), The relationship between JFace and SWT is most clearly demonstrated by looking at viewers and their relationship to SWT widgets, JFace provides viewers for most of the non-trivial widgets in SWT. Viewers are most commonly used for list, tree, table, and text widgets. Each viewer has an associated SWT widget. This widget can be created implicitly by supplying the parent **Composite** in a convenience viewer constructor, or explicitly by creating it first and supplying it to the viewer in its constructor

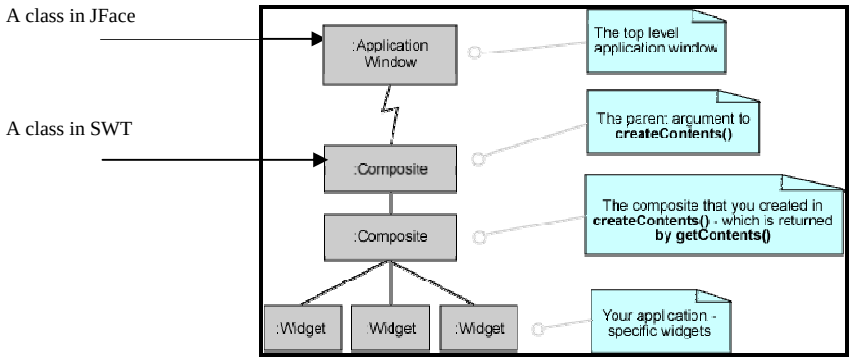

Figure 14. JFace Application Window relation to SWT wedgit

Lessons learned:

using clustering tools helps simplify the process of understanding systems with large repository , especially when no sufficient documentation is available for such systems, using Bunch clustering tool helps me understand two large systems, and enables me article the main differences between them, when I was not able to find any conceptual model or a complete documentation , that simplify the process of understanding such systems, and enables me to find how are the two systems related to each other.

What I found is , JFace is designed to provide common application UI function on top of the SWT library. JFace does not try to "hide" SWT or replace its function. It provides classes and interfaces that handle many of the common tasks associated with programming a dynamic UI using SWT.

The relationship between JFace and SWT is most clearly demonstrated by looking at viewers and their relationship to SWT widgets.

Using structured analysis tools helps answers questions about class dependences, which class affect which ?, what is the impact of change in the overall system, what I recognized by using this tool is that the improvement that JFace supplies SWT in the direction of simplifying GUI code, complicate the system and increase the amount of class dependences (coupling) , which made the process of any future modification of JFace System difficult , because it will affect many other classes, and that is considered one of bad design issue in the field of software engineering.

Figure 15 : The Listener cluster for the SWT

References:

1. The AT&T Labs Research website. http://www.research.att.com, last Access: Feb. 2006.

2. The Bunch Project. Drexel University Software Engineering Research Group
3. (SERG). http://serg.mcs.drexel.edu/bunch. last Access: Feb. 2006.

4. S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In Proceedings of International Conference of Software Maintenance, pages 50-59, August1999.

5. J. Korn, Y. Chen, and E. Koutso_os. Chava: Reverse engineering and tracking of java applets. In Proc. Working Conference on Reverse Engineering, October 1999.