Task-Based Visualization Using Merged View

Taeg Hyun Kang¹, Young Lee² and Mais Nijim²
1. Department of Computer Science, Texas Tech University, Lubbock TX 79409, USA
2. Department of Electrical Engineering and Computer Science, Texas A&M University-Kingsville, Kingsville TX 78363, USA

Received: November 01, 2011 / Accepted: January 12, 2012 / Published: June 30, 2012.

Abstract: This paper presents the concept of a merged view, task-based visualization technique, and it discusses the benefits and limits of merged view. The time and cost of understanding and analyzing the software are main factors of software development cost. Therefore, large and complex modern software systems need automated tools to aid programmers to understand the software system. This paper investigates how the merged-view guides a maintainer to locate the interconnected modules.

Key words: Software visualization, aspect oriented software development, software maintenance, merged view.

1. Introduction

Since software engineering field is opened up, maintenance of software systems has been one of the most important subjects AOSD (aspect oriented software development) is a paradigm that separates the crosscutting of concerns from the core concerns into modules called aspect [1, 2]. However, core concerns still problematic in a view of process for maintenance.

Software visualization is defined as a visualization of artifacts related to software and its development process [3, 4]. These artifacts include, design documents, modification of source code, and bug reports. Researchers in software visualization investigate new ways to graphically represent various aspects of software [5, 6]. Understanding the flow of interconnected source code in complex classes is the prime factor in software maintenance.

We accomplish this objective by temporarily merging the related source codes which are located in different files into a single view. Modification of source code in this single view is reflected in each separated files.

The proposed visualization tool, Java navigator (Jnavi), offers a new approach to understand and maintain software [7]. This shows the coupling of function and attribute among the classes at the time of maintenance. Jnavi improves the efficiency of programming by providing an interactive approach to maintenance. An associated code can be changed at the same time in different files. All codes which are related to the one specific code can be traced in a single view. This makes Jnavi a useful tool to understand and maintain software.

2. Jnavi

Jnavistore the java code in the database, analyze the relation between core concerns of classes and present all information specific core concerns which developers are try to modify. The overall system architecture of Jnavi is shown in Fig.1. The main components of the architecture are: (1) user interface, (2) SQL java code analyzer, (3) MySQL server. SQL java code analyzer syntactically analyzes java
source code which are stored in the database systems. User interface gives temporary merged view of related java source code.

Jnavi consists of two windows (Fig. 2). The left window is the navigator view and the right window is the editor view. Project and navigator window in Netbeans or Eclipse which are designed for java editor show the name of classes, methods and attributes in alphabetical order. To control the error occurrences, project view in Netbeans mark the class name with warning images. However, it is not sufficient to know the relationship between methods and attributes which are located in different classes. Empirically, we know that there are simple order for modification and development which shorten the time of modification.

Fig. 3 shows factory design pattern implement the coffee vending machine. This vending machine gives different type of coffee according to the inserted-coin.

![Coffee machine class diagram](image)

**Fig. 3** Coffee machine class diagram.
MainClass has core part that represents how to use the vending machine. To use the vending machine, we need vending machine object. Also, this vending machine object has a different type of coffee. To get a coffee, he/she has to insert the coin in the vending machine.

Afterward, the machine put into operation. Inheritance and invoking other object is important to understand the relation of classes. We can figure out whole operation of this factory pattern through the inheritance and invoking process. In this example, procedure of MainClass, MachineClass and WondooCoffee is the best approach to shorten the time when it takes until his understanding.

The Jnavi presents the order of the best approach with the temporary view. However, Netbeans only present related errors in compiling time.

3. Verifying Effectiveness of Jnavi

This chapter investigates the behaviors of developers during time of modification and total time from point of opening first java file to point of completion for modification. Many researchers in industrial engineering field do a motion study to improve predictability in the factory by minimizing the step of motion until whole process is finished. In the same way, we believe that we can improve the maintainability and predictability in developing the software systems by minimizing wasteful motion of developers. However, even though we minimize the motion of developers, if it is not guarantee the quality of software systems, it is useless. So we do a motion study in software engineering as well as research how Jnavi helps developers figure out the related core concerns by recording whole time for simple modification.

Fig. 4 shows the serial behaviors of developers in maintaining time. The developers repeatedly perform these actions. When request for modification occurs, the developers must reference the documents such as class diagram, sequence diagram to understand current status of software systems. Actual modification of source is more problematic. To modify and test the source code, developers must move around the editing view of the related classes.

Jnavi abbreviates these repeated steps during maintain time of software systems. To verify that Jnavi helps developers understand related core concerns precisely, we record the whole time until modification is done by using factory pattern of coffee vending machine. We request each developer to modify the results of this factory pattern classes.

Results of implementing this factory pattern are:
# your coffee is: Wondoo Coffee: blue Mountain
# your coffee is: Latte Coffee: caffe latte
# your coffee is: Cappuccino Coffee: moca

Modified results of this factory pattern are:
# your coffee is : Wondoo Coffee: I love blue Mountain
# your coffee is : Latte Coffee: I love caffe latte
# your coffee is : Cappuccino Coffee: I love moca

One group of developers uses the Netbeans, the other group of developers use the Jnavi to modify the crosscutting core concerns. Both groups of developers have less than 1 year experience in java.

Table 1 shows slight difference in total time. However, it is obvious that as software systems are bigger and more complicated, the gap between the results will be expanded. Moreover, this time gap will be accumulated due to the repetition of process, until the source code throws away.

In test for coffee machine example, the average time required in the Netbeans is 253.2 sec and it took 173.8 sec in Jnavi to modify the java source code (Fig. 5).
### Table 1  Total time for modifying (mi/ss).

<table>
<thead>
<tr>
<th></th>
<th>Netbean</th>
<th>Jnavi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2/58</td>
<td>3/23</td>
</tr>
<tr>
<td>2</td>
<td>3/05</td>
<td>4/43</td>
</tr>
<tr>
<td>3</td>
<td>2/41</td>
<td>4/03</td>
</tr>
<tr>
<td>4</td>
<td>3/01</td>
<td>3/56</td>
</tr>
<tr>
<td>5</td>
<td>2/44</td>
<td>5/01</td>
</tr>
</tbody>
</table>

Fig. 5  Required time comparison in Netbeans and Jnavi.

### 4. Conclusions

In java projects, shared classes which many developers refer should not be modified by one developer arbitrarily. Methods in the shared classes need to be blocked not to affect other classes and opened by the person who has the authorization.

We present Jnavi as a code visualization and maintenance tool. Through the design of Jnavi, we visualized the source code of core concerns as a flow of related code. This view led us to understand the codes which are related to the purpose of improvement without any additional documents such as sequence diagrams or class diagrams.

These features have two main benefits as follow:

First, developers repeatedly perform some actions such as “open related files”, “find other related functions” to understand the meaning of a code and check the progress of modification during the time of development. This tool saves time and cost by simplifying these motions of developers.

Second, frequent access to each file for modification disperses a developer’s attention. By merging the related source codes into a single view, developers can focus on the purpose of improvement.

### References


