Chapter 1 Introduction

1.1 Motivation

“Thirteen, this is Houston.”

“We're reading your telemetry.”

“It's good to see you again.”

“Good to see you, too, Houston.”

This is a part of movie script in the movie “Apollo 13.” In the movie, the space shuttle has trouble in space. They got to solve all of problems in order to come back to earth. They use the control system to operate the space shuttle and monitor all its status. When any problem happens in space shuttle, control system will respond to give an alarm from control panel to show spaceman which part of system is in trouble. For example, in the movie, if engine five malfunctioned then the alarm LED of engine five will be switched on to show engine five is in trouble. If CO2 filter doesn’t work then CO2 detection system will signal the spaceman. Spaceman could understand what happens in space shuttle easily by the alarm message and report the problem to Houston in a minute.

Can software nowadays does the similar alarm scenario as in “Apollo 13?” If any

exception happens in software, could people know which part of the software is broken, like the control panel of “Apollo 13” which shows all the status of system. So, the objective of this paper is when software exception occurs, we could display a component map to show the architecture of the program and illustrate us which component may cause the exception.

The software is a black box to end user. The software nowadays can allow users to configure which component to install, such as plug-in, etc... But, after such initial configurations, the software remains as an abstract black box to end user, especially when a program crashes. For example, in Windows environment when a program crashes, Windows usually show a message box to require end-user to report back. It reports information about the crashed program like memory dump, call stack, and etc. Using the information, the software company could improve their product in the next version. However, it provides no help to end user. First, end user might continue executing the program and meet the same error then crash again. Second, even crash information reaches software developers, end user must wait for the next version or patches that fix the problem in a long time.

The goal of our research is trying to propose a solution to this problem. Take hardware for example, if a personal computer does not work, we can diagnose which part is malfunctioned to some extent and replace it with parts from the market. On the

other hand, if our software does not work, we can not diagnose which component is in trouble.

To approach this goal, we need to solve several technical issues. First, we use Java error handling to catch exception. The architecture of software is composed by components. When a Java application run into an exception, our system catch it , displays a component map to show the relation of all components inside the application, and then provide diagnosis information of which component may cause the exception.

Although we could catch an exception from a component, often the root cause is not from it. So, we choose to analyze the source code using program slicing. Using program slicing, we could get relation of all components inside a program, conjecture the root cause of the exception, and then provide essential diagnosis information in the component map.

Program slicing is a method used by in software engineering community for abstracting programs. The static slice is the set of all statements that might affect the value of a given variable occurrence. Program slicing is useful for debugging, because it potentially allows one to ignore many statements in the process of localizing a bug. If a program computes an erroneous value for a variable x, only the statements in the slice w. r. t. x have (possibly) contributed to computation of that value; all statements

which are not in the slice can safely be ignored. Before making a change to *s*, one could examine the slice *s*, indicating the program part affected by *s*. this may produce useful insights how the error may be corrected.

When program crashes, we catch the exception and get the number of line where it crashes. However, we are not sure where the root cause is. So, we need to use program slicing to diagnose which statments may cause the exception and then locate which component it belongs to. Thus we could infer the statements which could causes the exception by using program slicing and show this diagnosis information in component map.

Our thesis is organized as follows: In Chapter 2, we provide the background starting from basis of Java technology that we use (includes class loader, handing error with exception and Java access bridge), software component Java bean, program slicing (includes static slicing and dynamic slicing), Parse builder tool Programar and graphic API: Dot. In Chapter 3, we give an overview of our system architecture and illustrate how we execute a Java application and monitor it in our system. In Chapter 4, we describe our analysis implemented by using program slicing techniques. In Chapter 5, we describe how we diagnose an exception.