# Chapter 3 Intercepting the exceptions

## 3.1 System architecture

Our system's function is executing a Java application, catching its exceptions, displaying the relation of all components inside the Java application, and showing which component may cause of the exception. According to these functions, we divided out system into four parts, where is shown in Figure 3-1. "System manager" is used to control the system and it is also the user interface of the system. *JavaAp_Loader* is used to execute a Java application. "Exception diagnosis system" is responsible of providing clues of which components the root crash cause may locate "Graphic Displayer" could create a graphic file and display it. This is the typical work flow of system:

1. Execute a Java application file:

   User input a Java class file to our system. System executes it and monitors it.

2. Catch the exception if it happens:

   Our system catches the exception if any exception occurs. Our system could

get the stack trace from the exception information and then initiate "Exception diagnosis system" to analyze the cause of exception.

3. Analyze the cause of exception:

   We could conjecture that the cause of exception, according to the stack trace and the information computed from program slicing. Finally, we call the "Graphic Displayer" to create the picture file (dot file) according to the result of analysis.

4. Display the cause of exception using graphic:

   When "Graphic Displayer" is called by "Exception Diagnosis System", it creates a picture file that we call "component map" to show the architecture of the application and show the diagnosis information from the analysis.

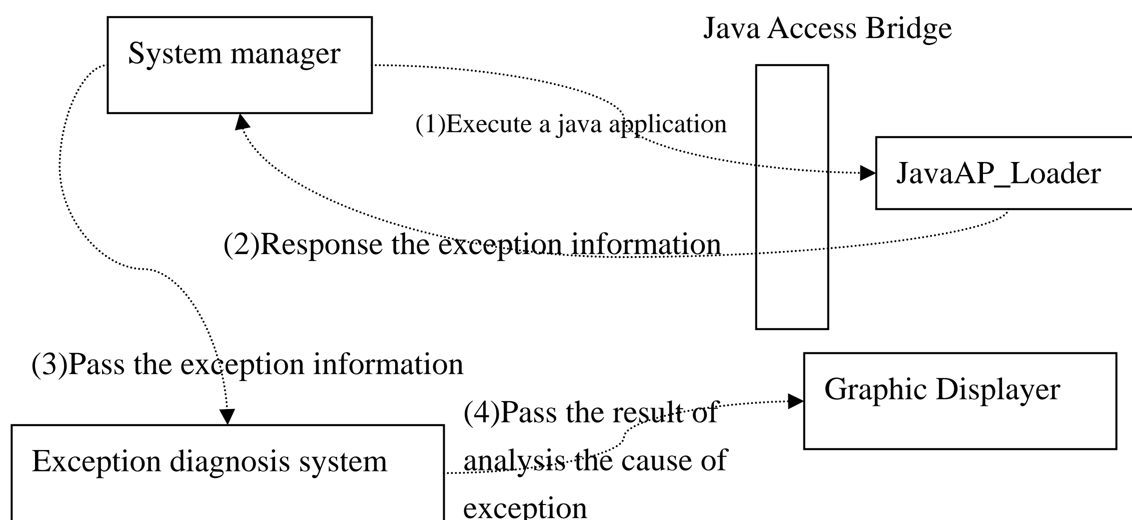


Figure 3-1 : Interaction of the four parts inside the system

# 3.2 The design of the java application loader

*JavaAP_Loader* is used to execute a Java application , monitor, and catch the exception. When *JavaAP_Loader* executes a Java application, it responds two kinds of results, “Termination” and “Exception”. ”Termaination” means a Java application terminate normally. If the result is an exception, *JavaAP_Loader* get the stack traces from the exception information and return them to system manager. Figure 3-2 expresses four states of *JavaAP_Loader.*

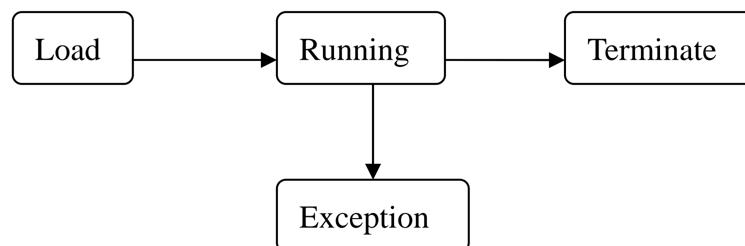


Figure 3-2: The status of *JavaAP_Loader* when executing a java application

Recall that our system is based on Windows platform. So, we use MFC (Microsoft Foundation Class) to develop the system. However, the *JavaAP_Loader* is

a Java application. Communication between a Java application and normal program is much more complicated. Java provides a program called Access Bridge to allow Java application to communication with Windows application. We could monitor a Java application's states (Figure 3-3) when is executed by *JavaAP_Loader*. Figure 3-3 illustrates interaction between System manager (written in C++) and *JavaAP_Loader*. System manager communicates with *JavaAP_Loader* by Access Bridge, including executing a java application and get the stack trace generated by *JavaAP_Loader*.

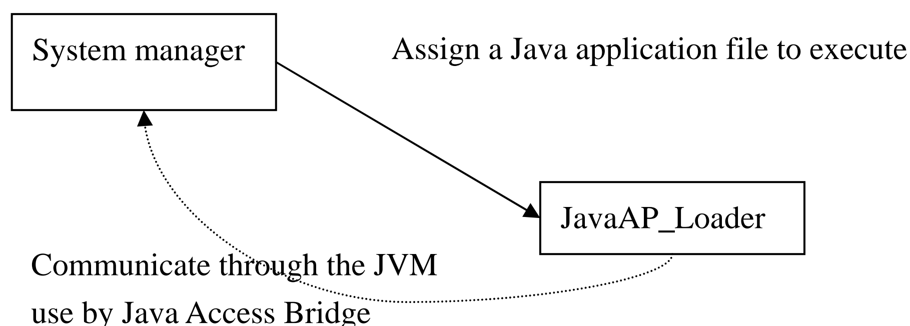


Figure 3-3: Interaction and communicati (stack traces) between system manager and ***JavaAP_Loader*** if exception happens

# 3.3 Monitoring and catching the exceptions

## 3.3.1 An overview of Java class loader

Java class file are not executable directly: it is loaded into JVM and executed by

JVM. Java provides a class called "*ClassLoader*" to be overwritten. The class *ClassLoader* is an abstract class. Applications can implement subclasses of ClassLoader in order to extend the manner in which the Java Virtual Machine dynamically loads classes. So, if we want to monitor the execution and loading of Java application we want to implement a proprietary *ClassLoader.*

### 3.3.2 The implementation of a proprietary class loader

A *ClassLoader's* basic purpose is to serve a request from a class. When JVM needs a class, it asks the *ClassLoader* by class name. If the requested class is already loaded by JVM before, the *ClassLoader* should return that Class object. If the requested class file has never been loaded, *ClassLoader* by default searches the class files along the environment variable CLASSPATH over the file system. This is fine for regular situations, when you have your Java program fully compiled and waiting on your computer. But one of the most innovative features about the Java language is that it makes it easy for the JVM to get classes from places other than the local hard drive or network. For example, browsers use a custom *ClassLoader* to load executable content from a Web site.

Java APIs provide *URLClassLoader* that we could use it to customize a

ClassLoader. This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories. Any URL that ends with a '/' is assumed to refer to a directory. Otherwise, the URL is assumed to refer to a JAR file which will be opened as needed. *URLClassLoader* provide a method that is used to append the class path that we need to the list of URLs to search for classes and resources.

The class path tells SDK tools and applications where to find third-party and user-defined classes -- that is, classes that are not Java extensions or part of the Java platform. The class path is the directive for finding any classes you've compiled with the javac compiler -- its default is the current directory to conveniently enable those classes to be found.

Java 2 SDK, the JVM and other SDK tools find classes by searching the Java platform (bootstrap) classes, any extension classes, and the class path, in that order. Class libraries for most applications will want to take advantage of the extensions mechanism. We only need to set the class path when we want to load a class that's (a) not in the current directory or in any of its subdirectories, and (b) not in a location specified by the extensions mechanism.

#### 3.3.2.1 Execute a class

After implement a class loader, we could get a class object associated with the class or interface with the given string name, using our class loader. In a normal situation, JVM executes a Java application from the “main” method of the class which is passed as a parameter to the JVM (java). However, since our class loader needs to execute a Java application by ourself, we need to emulate the work which is done by JVM.

In Java, there is a special class called *Class*. The *Class* class provides a method that could return a *Method* object that reflects the specified declared method of the class or interface represented by the *Class* object. We could use the method to get a *Method* object. A *Method* object provides information about, and access to, a single method on a class or interface. It permits widening conversions to occur when matching the actual parameters to invoke with the underlying method's formal parameters. So we could get a *method* object that named “main” from the *Class* object and then invokes the underlying method represented by this *Method* object, on the specified object with the specified parameters.

## 3.3.3 Catch the exception

### 3.3.3.1 Get error handle

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. So we could execute a class within a try block as in Figure 3-4. When exception happens, we could catch it in our system.

```
try {
Execute the java application
}catch(Exception e) {
Catch the exception and respond stack trace
}
```

Figure 3-4 : Get the exception that the application is running in out system

### 3.3.3.2 Get stack trace

The class Exception and its subclasses are a form of *Throwable*. The *Throwable* class is the superclass of all errors and exceptions in the Java language. A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a *cause*: another throwable that caused this throwable to get thrown.

When we catch an exception, we need the *Throwable* object of the exception to get the stack trace information. In Figure 3-5, that is a stack trace when exception happens.

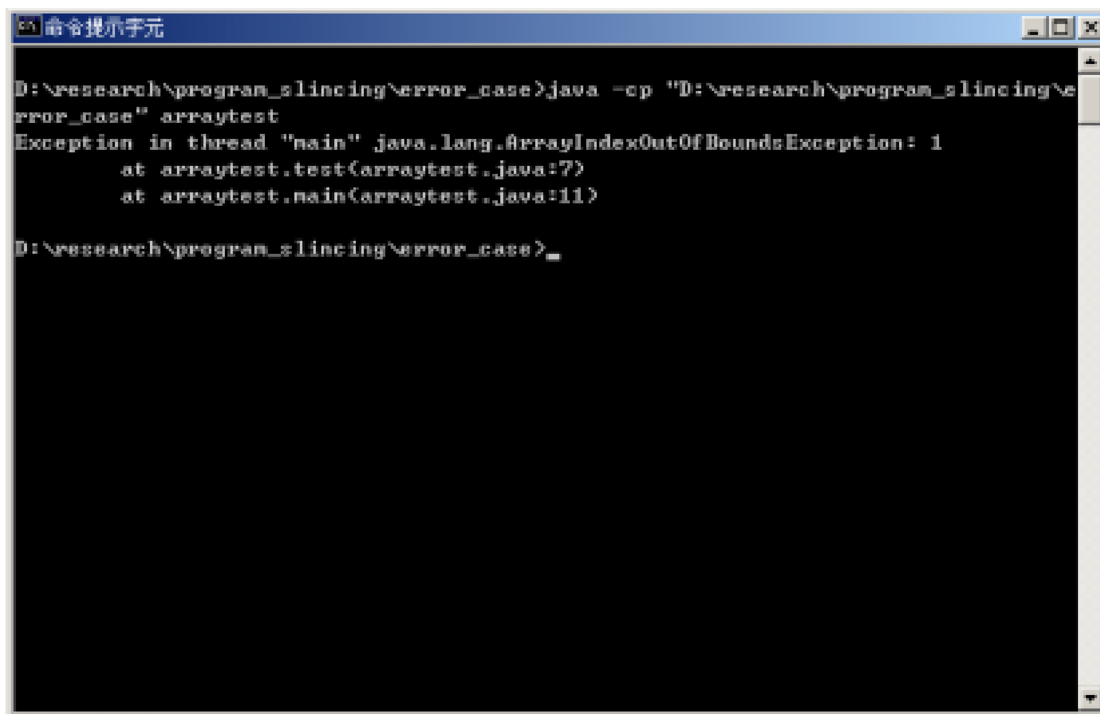


Figure 3-5 : show the stack trace when exception happens

The information of stack trace is an array of elements, each representing one stack frame. The first element of the array (assuming the array's length is non-zero) represents the top of the stack, which is the last method invocation in the sequence. Typically, this is the point at which this throwable was created and thrown. The last element of the array (assuming the array's length is non-zero) represents the bottom of the stack, which is the first method invocation in the sequence.