ASPECT-ORIENTED EXCEPTION HANDLING FOR NETWORK PROGRAMMING IN A COMPONENT-BASED FRAMEWORK

Takeru Ujinawa  
Graduate School of Frontier Informatics,  
Kyoto Sangyo University  
Kyoto, Japan  
email:

Naohiro Hayashibara  
Faculty of Computer Science and Engineering,  
Kyoto Sangyo University  
Kyoto, Japan  
email: naohaya@cse.kyoto-su.ac.jp

ABSTRACT
Exception handling is one of the most important factors in software development. It provides fault-tolerant functionality for possible errors occurred during execution. Most of modern programming languages (e.g., Java, C++, OCaml, Ruby, etc.) have such function. On the other hand, it is able to make the program complicated. One of the main drawbacks is to generate code clones.

In this paper, we focus on exception handling on networking using aspect-oriented approach in a component-based framework implemented in Java. Handling I/O related exceptions is separated from the main code and then weave it into the program before compiling. It reduces the cost of programming and increases readability of the program code. This approach is implemented as an extension of the Squareknot framework. We also discuss on the effect of this approach in the view point of source code and bytecode.

KEY WORDS
Exception handling, aspect-oriented programming, component-based framework, network programming

1 Introduction
In the case of programming using communication function, developers are often required to handle exceptions by the specification of modern programming languages (e.g., Java, C++, OCaml, Ruby, etc.). It is important for avoiding unexpected error by failure of hardwares, an operating system or unexpected input during execution.

On the other hand, it often makes the program complicated. One of the main drawbacks is to generate code clones. In the case of Java, IOException is for example required to be handled in a code (i.e., required throws or catch on the exception) by the language specification, otherwise the code fails to compile. However, handling this type of exceptions is often realized by almost the same code. It means that there can be many code clones for handling exceptions when the program includes various operations on the network. It is of course reason to degrade its readability.

In this paper, we focus on a Java-based programming framework, called Squareknot [1]. It is a component-based framework to connect various controllers with various actuators (e.g., networked robots, electrical appliances, etc.) by using various types of network (e.g., TCP/IP, Bluetooth, ZigBee, etc.). You can operate a desired actuator with any controller via Squareknot if they have network connection capability. The framework is composed of modules and the mapping engine. Each module in the framework is related with each device (i.e., controller or actuator). The mapping engine takes the role of interrelating a controller module with an actuator module and then the related actuator can be operated by the controller. Although each module uses various libraries to connect an actual device, it can be co-existent with other modules because it is independent from the others.

To use various libraries developers have to take account of exceptions, not only in the standard library but in external libraries when they implement modules in the framework. Specially, exception handling in communication (e.g., socket generation, operations on a stream, etc.) cannot be omitted, thus we focus on communication related exception handling.

Our solution is to separate exception handling on network I/O (especially, connection and disconnection) from the main code to improve readability and reduce mistakes (e.g., forget to catch or throw exceptions) then weave it into the code subsequently. It means that developers do not consider about such exceptions basically, and consequently reduces the cost of implementation. It is realized by aspect-oriented programming paradigm. This approach is implemented as an extension of the Squareknot framework.

The rest of the paper consists of the following sections. We show some of related works in Sec. 2. We then introduce about the Squareknot framework which we use in this work in Sec. 3 and describe our approach on aspect-oriented exception handling on Squareknot framework in Sec. 4. We also evaluate the approach in various view points in Sec. 5. Finally we summarize the work in Sec. 6.

2 Related Work
There exist various research works using aspect-oriented programming paradigm for the last several decades.

Lippert and Lopes [2] discussed about exception
detection and handling using aspect-oriented programing paradigm (AOP). They revealed that AOP can support for reducing the number of coding steps by separating the main procedure from the code on exception handling. It has proven the advantages of the implementation with JWAM and the evaluation on it. It also contributes to improve tolerance for changes in the specification. On the other hand, they pointed out several weakness of AOP, for example, conflicting semantics of AOP and Java language, insufficient support for reconstruction of the local effects.

Extension of AspectJ has been discussed in [3]. They propose the simple extension to AspectJ called explicit join points (EJPs). AspectJ can increase readability in Java source code. Because it can separate crosscutting concerns of some objects. By contrast, AspectJ has several deficiency (e.g., the flow is difficult to trace.). EJP can detect potential occurrences of aspects of the base code. Additionally it can be coupling the aspect with base code.

Souchon et al. proposed the way of exception handling in component-based system [4]. In exception handling of component-based system has serious problem, for example, when exception had occurred in using asynchronous communications. They categorize these problems into three, CS components (contract-based interactions using synchronous communications), CA components (contract-based interactions using asynchronous communications) and E components (event-based interactions using synchronous communications). Besides, they propose several EHSs (exception handling systems).

AutoFlow [5] is an another approach to support AspectJ. Processing sequence is difficult to trace in source code using AOP. Thus, many bugs remain source codes and developer need to debug source codes. AutoFlow is automatic debugging tool for aspectJ software. It shows the warning point using change impact analysis. Therefore developer can notice problems in aspect source code.

Molnar et al. proposed a function trap tool using AspectJ [6]. The tool weaves an aspect to specific method calls (e.g., setter method, getter method, etc.). These aspects are associated with a log method which generates a log file in XML format. The information described in the log file is more readable than normal log files because it can express, for example, date of execution, method signature, parameters in the tree structure. Also it has a graphical user interface to enhance the readability of log files.

Accordingly, Multilevel Security (MLS) is necessary to improve the security level. Kotrappa and Kulkarni developed the approach to implement the procedure on authentication as an aspect and then the aspect is inserted into the program by using AspectJ [7]. Although the proposed approach can be used for the development of security softwares in general, they implemented MLS with security requirements in jFTPd using the approach. This MLS utilizes BLP (Bell-LaPadula) model for classifying users into several authority. In this case, aspect can weave several level security modules.

3 Squareknot: A Component-based Programming Framework

In the previous work, we’ve developed a component-based framework, called Squareknot implemented in Java [1]. It is for connecting a controller with any actuators (e.g., robots, sensors, electrical appliances, etc.) by using various type of network (e.g., TCP/IP, Bluetooth, ZigBee, etc.).

3.1 Overview of the framework

This framework is composed of modules, the mapping engine and communication layers (see Fig. 1). It has two types of modules, one is modules for controllers, called controller modules, the other is for actuators, called actuator modules. Each module is corresponding to a real device via network.

![Figure 1. The architecture of Squareknot](image)

We now explain the components in the Squareknot framework briefly.

Controller modules

This type of modules corresponds to a controller device to receive and understand the input from the device.

Actuator modules

This type of modules corresponds to an actuation device to translate a set of operation from a controller device and send it to the device.

Mapping engine

It takes on the role that associates a controller module with an actuation module according to user requirement.

3.2 Module system

Each module is also detached from other modules in the sense of method calling. It is very important feature to
realize MAC (mapping between actuators and controllers) model [1]. Mapping between controllers and actuators should be flexible according to user’s requirement. Someone may require to use a controller for operating various devices. In this case, the framework should change the connection between a controller and actuators. In the Squareknot module system, a module is an object which includes methods for operations and socket for connecting to a real device.

Controller module is a passive object which receives some operation on the controller device and actuator module translates the operation into the appropriate format and sends it to the corresponding actuation device. Thus, the framework just changes the connection between controller modules and actuator modules logically when users require to change it on real devices.

Modules in Squareknot are pluggable and hence it is very easy to import a new device into the framework. In this case, developers do not take care of the internal relationship of the framework.

### 3.2.1 Communication with devices

Modules use various types of network to connect with real devices (i.e., controllers and actuators). Thus, we cannot avoid network related programming for developing modules in Squareknot. Specially, developers are required to handle I/O-related exceptions to avoid serious problems during execution. It can often be an obstacle to implement a module when we use various types of network because, for example, exception handling is different from Ethernet when we use Bluetooth to communicate with devices. It may also be trouble by libraries which we use for the implementation. In Squareknot, every module uses some libraries to communicate with the corresponding device. Thus, this problem is so much conspicuous.

In most of modern languages, handling exceptions related to network is necessary in the view point of security and reliability of softwares. However, the cost of programming must be reduced if developers do not care about this problem so much. This work that concerns to exception handling started from the problem.

### 3.3 Mapping engine

The mapping engine takes the role that manages pairs of a controller module and an actuator module. When a user push the Mapping button on choosing a controller and an actuator on the GUI shown in Fig. 2, it makes a connection for both the controller and the actuator by using modules.

Then, control information for the actuator is given by the controller intermittently. It keeps connecting on them until the user pushes the Disconnect button (see Fig. 2).

The mapping engine can also change the connection between a controller and an actuator dynamically. It quickly changes the relationship between modules when the user requires to change the target controller or actuator.

### 3.3.1 Configuration files

It still remains the problem of mapping operations on a controller with actions on an actuator because there are many ways to operate the actuator with the controller. However, we need to define the unique action for each button or action on the controller.

We solved the problem with configuration files (see. Fig. 1). It defines the relationship between the operations on a controller and the actions on an actuator. It allows users to make fine-grain customization for operating the actuator by the controller without modifying the source code of the modules. Configuration files can be selected in the another window located in the bottom in the GUI (see Fig. 2).

For the same pair of a controller and an actuator, there can be several ways to control the actuator by the controller by using different configuration files.

### 3.4 Features of the framework

Squareknot has three major features.

**Easy startup and intuitive operation**

Squareknot has some controller modules (e.g., WiRemoteController, WiiBalanceBoard) and actuator modules (LegoMindstorms, AR.Drone, DebugMode). Thus, you can use this framework easily.

And, if you execute the jar file, you can use Squareknot with simple GUI.

**Extensibility**

There is no relationship between modules in Squareknot. Thereby several modules have no dependency
with other modules. As a result, you can install and remove desired module anew like plugin.

**Flexibility**

Operational mapping between module is easy. This arises from simply configuration in configuration file. Configuration files are written with text format. Therefore you are rewritten simply.

And you can choose the connection of the modules from GUI.

### 3.5 Required environment

We have tested the Squareknot framework implemented the proposed approach on the environment shown in Table 1.

<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 13.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM</td>
<td>JVM 1.7.0_21</td>
</tr>
<tr>
<td>Framework</td>
<td>Squareknot-0.22</td>
</tr>
</tbody>
</table>

Table 1. Required environment for Squareknot.

Modules in Squareknot are using the libraries listed in Table 2.

<table>
<thead>
<tr>
<th>For Bluetooth</th>
<th>bluecove ver. 2.1.1 [8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Wii remote</td>
<td>WiiremoteJ ver. 1.7 β [9]</td>
</tr>
<tr>
<td>For LegoMindstorms</td>
<td>lejos-nxj ver. 0.9.1 β [10]</td>
</tr>
<tr>
<td>For AR.Drone</td>
<td>JavaDrone1.3 [11]</td>
</tr>
<tr>
<td>For Configuration files</td>
<td>Apache commons-conf. ver. 1.6</td>
</tr>
</tbody>
</table>

Table 2. Java libraries used by modules in Squareknot.

### 4 Aspect-Oriented Exception Handling

Exception handling is required in modern programming languages especially on network I/O. Thus, it is also important when each module connects with a corresponding device in Squareknot. Developers have to take account of specific exceptions defined in libraries that are used in modules. It is one of the reasons that the programming cost is increasing. We propose the aspect-oriented exception handling for network programming.

#### 4.1 Aspect-oriented programming

Aspect-oriented programming (AOP) is a well-known programming paradigm to solve the problem of object-oriented programming (OOP). OOP cannot handle very well about code frequently and transversally used in the whole program (e.g., logging, etc.). It has often been viewed as a problem of code clones. AOP can separate from this type of code from the main procedure as an aspect and then put it into the program as necessary before compiling (i.e., weaving).

One of the implementation for aspect-oriented programming in Java is AspectJ [12].

The mechanism of aspect weaving is very simple in AspectJ. You can describe some code, called an advice, which is put into the program as an aspect. Then you express a signature of the method, called a join point, where you would like to weave the aspect. Timing of the execution of the advice at the join point can be specified by using the following keywords in AspectJ.

**before**

The advice is executed before the method execution.

**after**

The advice is executed after the method execution.

**after returning**

The advice is executed after the method returned.

**after throwing**

The advice is executed after exception handling in the method.

**around**

The advice is executed before and after the method execution.

Several join points can be collected as a pointcut to weave the same advice. There are two types of pointcuts: call and execution.

**call**

In the case of weaving with a pointcut defined using the keyword call, the corresponding advice is inserted into the code in the method caller. Invocation using RMI (remote method invocation) and reflection in Java is not identified as the normal method call by using AspectJ. Thus, the advice is not inserted using the weaving of AspectJ in this case.

**execution**

In the case of weaving with a pointcut defined using the keyword execution, the corresponding advice is inserted into the inside of the method invoked. Therefore the advice can be woven into the method when is invoked using RMI or reflection.

The logging sample code of AspectJ is shown below.

```java
public aspect SetterInterceptor {
  pointcut logging():call(void set());
  before(): logging() {
    log.info("--setter is called--");
  }
}
```

Program 1. Example of the aspect including a pointcut and an advice.
In this case, the method `logging()` is executed before calling the method `set()` by weaving.

### 4.2 Handling exceptions for network programming

In this paper, we propose the way of exception handling for network programming. Specifically, exception handling is mainly required for protecting the program when it communicates with external devices because it could have unexpected input from them during its execution.

To handle exceptions, we have to write almost the same code in several places in the program. Specially, we need to describe numerous codes for exception handling on networking in Squareknot because modules connect to controller and actuation devices with various network protocols (e.g., TCP/IP, Bluetooth, ZigBee, etc.). It might be an obstacle for developers to implement modules with these protocols because the way of exception handling is different according to the implementation (i.e., library) to use the protocol.

Now we focus on the exceptions concerning to connecting and disconnecting operations for remote devices because these operations are frequently used in the framework. Our approach separates the code of handling these types of exceptions from the main procedure and then weave it into the program of each module before compiling.

We express such exception handling as an aspect and weave it into the program by AspectJ.

We introduce the detail of the implementation of the aspect-oriented exception handling in the following section.

### 4.3 Implementation

Exceptions which concern to establish a connection or its disconnection are a family of `IOException`. Thus, such exceptions are handled as `IOException` in our approach. On the other hand, code for handling such exceptions is separated as a set of aspect code and then it is woven by AspectJ.

To derive the benefit, developers are required to define the method signature, `void connect(void)` for connecting operation, `void disconnect(void)` for disconnecting operation, in controller modules and actuator modules in Squareknot. Developers have to keep the coding rule if they want to be lazy in handling these exceptions.

#### 4.3.1 Advantage of the proposed approach

Practically speaking, various modules use various libraries to communicate with controller and actuation devices with

```
public aspect ExceptionInterceptor {
  pointcut connectException():execution(* *.connect(..));
  pointcut disconnectException():execution (* *.disconnect(..));
  after() throwing(IOException e) :
    connectException() {
      e.printStackTrace();
      System.out.println("------IOException in connect------");
    }
}
```

Program 2. The aspect for handling exceptions concerning to connection and disconnection

```
public void connect() throws IOException{
  //Connection procedure
}
```

```
public void disconnect() throws IOException{
  //Disconnection procedure
}
```

Program 3. Example of source code of a module

In Program 2, we show the aspect including pointcuts and advices for weaving the code of the modules. Exception handling in the methods, `connect()` and `disconnect()`, has been done in the same manner according to the aspect. If you wish to describe an additional procedure in these methods, you need to add it into the aspect.

As we mentioned in Sec. 4.1, there are two kinds of join points, `call` and `execution` in AspectJ. We need to use them for weaving the advices on exception handling shown in Program 2 into the code in modules appropriately. The mapping engine invokes the `connect` and `disconnect` method in each module (a class in Java) by reflection. It is important for dealing with various modules in the same way. It means that these advices have to be inserted in the code of each module for dealing with exceptions. Thus, we chose the `execution` join point to weave the advices into the code for each module (see line 4 and 5 in Program 2).

4.3.1 Advantage of the proposed approach

Practically speaking, various modules use various libraries to communicate with controller and actuation devices with

```
import java.io.IOException;
public aspect ExceptionInterceptor {
  pointcut connectException():execution(* *.connect(..));
  pointcut disconnectException():execution (* *.disconnect(..));
  after() throwing(IOException e) :
    connectException() {
      e.printStackTrace();
      System.out.println("------IOException in connect------");
    }
}
many kinds of network protocols. Such libraries which are used in modules includes some specific exceptions to deal with some serious error. Therefore, developers always consider about the details of many libraries.

In our approach, the code would be simple because the common exception handling related to connecting and disconnecting operations is separated from it. Of course, you can add some procedures for error handling in the aspect like Program 2 if you need. Then the procedures are inserted into the modules before compiling.

In the case that you wish to add a certain procedure for exception handling on the specific error, you can write it into the module directly.

4.3.2 Remaining issues

In our approach, exceptions related to connection and disconnection are handled as the IOException. However, this type of the exception is widely used for handling input/output error with external devices. Sometimes, this fact might be a reason that bug tracking is complicated.

To solve this problem, we are considering to define a new exception class to aggregate exceptions concerning with network I/O and implement it in the future.

5 Evaluation

Exception handling is essentially a different thing from the main procedure. Thus, our approach increases the readability of source code by separating some kind of exceptions. Moreover we’ve compared two frameworks, one includes our approach proposed in this paper and the other does not, in the view point of source code and bytecode.

5.1 Scenario of the evaluation

We evaluate the proposed approach to compare it with the existing implementation of Squareknot in the view point of source code and bytecode.

There is an evaluation of handling communication-specific exceptions. We evaluate this work by two approaches, source code and bytecode. In order to evaluate this work, we implemented two types of codes with and without the proposed approach for handling communication-specific exceptions to Squareknot framework.

Squareknot framework has some modules, controllers and actuators. Moreover modules communicate with other modules by several protocol (e.g., TCP/IP, Bluetooth). Features of each and aspect are as follows.

5.1.1 Controller modules

We picked the following two modules for the comparison.

WiiBalanceBoard

A module is to use Wii balance board (Nintendo). It can be operated by weight movement on the board. The module uses Bluetooth with HID profile to communicate with the device. We also use WiiRemoteJ library [9] for communicating with Wii Balance Board.

WiiRemoteController

A module is to use Wii remote controller (Nintendo). It is a game controller including buttons and an acceleration sensor. The module uses Bluetooth with HID profile to communicate with the device. We also use WiiRemoteJ library [9] for communicating Wii Remote Controller.

5.2 Comparison on source code

Evaluation did make use of two types of codes, non-aspect code and aspect-embedded code. Aspect is woven with only "connect" method.

```java
public void connect(){
    System.setProperty("bluecove.jsr82.
    psm_minimum_off", "true");
    try {
        while(aWiiRemote == null){
            aWiiRemote = WiiRemoteJ.findRemote();
        }
        aWiiRemote.addWiiRemoteListener(this);
        boolean[] lights = { true,false,false,
            false};
        aWiiRemote.setLEDLights(lights);
        System.out.println("Connect to Wii!");
    } catch (IllegalStateException e) {
        e.printStackTrace();
        System.out.println("Error1");
        aWiiRemote.disconnect();
    } catch (InterruptedException e) {
        e.printStackTrace();
        System.out.println("Error2");
        aWiiRemote.disconnect();
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error3");
        aWiiRemote.disconnect();
    }
}
```
Program 4. connect method in the controller module for Wii remote in the existing approach.

```java
public void connect() throws IOException{
    System.setProperty("bluecove.jsr82.
    psm_minimum_off", "true");
    while(aWiiRemote == null){
        aWiiRemote = WiiRemoteJ.findRemote();
    }
    aWiiRemote.addWiiRemoteListener(this);
    boolean[] lights = {true,false,false,
    false};
    aWiiRemote.setLEDLights(lights);
    System.out.println("Connect to Wii!");
}
```

Program 5. connect method in the controller module for Wii remote in the proposed approach.

In Program 4, exception handling occupies half of the connect method. Thus it is difficult to read the source code. In contrast, if you use proposed approach, necessary instructions are described to the source code. Hence readability increases like Program 5.

<table>
<thead>
<tr>
<th>Module</th>
<th>normal(lines)</th>
<th>proposed(lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WiiBalanceBoard</td>
<td>146</td>
<td>138</td>
</tr>
<tr>
<td>WiiRemoteController</td>
<td>238</td>
<td>230</td>
</tr>
<tr>
<td>ArDrone</td>
<td>130</td>
<td>122</td>
</tr>
<tr>
<td>LegoMindStorms</td>
<td>112</td>
<td>102</td>
</tr>
</tbody>
</table>

Table 3. Comparison on source code

Program 4 is an implementation of connect method in the module for Wii remote controller. It is implemented in the existing framework straightforwardly. It is necessary to put the code for exception handling into the program. In this case, most of the code are spent for exception handling.

On the other hand, Program 5 shows the implementation of the module in the proposed approach. It increases readability by separating the exception handling from the main procedure. The aspect for handling exceptions is automatically inserted into the module if developers forgot handling the exceptions. It is a kind of fail-safe function on connection and disconnection over the network in the Squareknot framework.

5.3 Comparison on bytecode

In Java bytecode, each line corresponds to an action (instruction step) in Java virtual machine. Thus, execution time on Java virtual machine is proportional to the number of instruction steps in general. We compare with both the existing one and the proposed approach on the number of instructions of bytecode and show that there is no performance hit in the proposed approach from the result.

Table 4 shows the number of instruction steps in bytecode on both approaches, the existing implementation and the proposed one, for each module.

<table>
<thead>
<tr>
<th>classname</th>
<th>normal (steps)</th>
<th>proposed (steps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WiiBalanceBoard</td>
<td>308</td>
<td>309</td>
</tr>
<tr>
<td>WiiRemoteController</td>
<td>560</td>
<td>549</td>
</tr>
<tr>
<td>ArDrone</td>
<td>192</td>
<td>197</td>
</tr>
<tr>
<td>LegoMindStorms</td>
<td>148</td>
<td>149</td>
</tr>
</tbody>
</table>

Table 4. Comparison on bytecode

According to the result shown in Table 4, there is no observation on a big increment of the number of steps. It shows that the proposed approach does not give additional performance hit on the execution time while it gives the readability of the code and reduces the cost of implementation.

6 Conclusion

We have shown our approach to exception handling on networking using aspect-oriented programming paradigm. It is implemented as an extension of the Squareknot framework. In Squareknot, modules often use connecting and disconnecting operation to communicate with remote devices over the network. Thus, exception handling for such operations is required to preserve the framework from unexpected interaction with external devices.

Our approach focuses on exceptions that have a concern in connection and disconnection on the network and separate the main procedure of each module from the code handling such exceptions. Exception handling is described as an aspect which includes the error handling code and the points where the code is inserted into. Then, the framework weaves the exception handling procedure into appointed places in modules by using AspectJ.

We also discussed about the proposed approach in the view point of source code and bytecode. There are lots of lines for exception handling in each modules in the existing approach, nevertheless they are common things. It means there are lots of code clone widely spreading across modules in Squareknot (e.g., the code in Program 5 in each module).

So, we aggregated the common exception handling procedure on connecting and disconnecting operation as the aspect shown in Program 2 and weave it into the code in each module appropriately. Therefore, the exception handling code which is described as advices beforehand is inserted before compiling when developers forget writing such a code.

The proposed approach is able to reduce the cost of programming and to increase the readability of the code of
modules in Squareknot without the additional performance hit. We are trying to aggregate some other exceptions related to network (e.g., operations on a stream, etc.) in the future work.

Acknowledgements

We are grateful to Prof. Atsushi Aoki for his insightful comment which improved the manuscript. This work was supported by MEXT/JSPS KAKENHI Grant (Grants-in-Aid for Young Scientists (B)) Number 24700078.

References


