

Using a Declarative Chain of Responsibility Pattern to Write Robust, Self-Correcting Distributed Applications

Dan Stieglitz
Principal Consultant
Stieglitech, LLC
dan@stieglitech.com

Abstract

Businesses are in a constant struggle to improve application reliability, a fight that is made more difficult by the increasing complexity of the business application environment. New distributed components, databases and other moving parts are added to enterprise systems in a never-ending stream. Many application architects embed application-level fail over into applications in an ad hoc manner, which leads to compatibility and consistency issues for current and future developers. This paper presents a strategy for building a more easily maintainable, componentized exception handling subsystem using a declarative chain of responsibility pattern.

Introduction

Subsystems specifically designed for exception handling (an aspect found in every project) can be strangely rare in the business world. There are few if any vendor products available that are “generic, rich-featured exception handling suites.” Furthermore, when exception handling frameworks are home-built, they are construed to contain much difficult logic and to be fraught with peril. There seems to be a wide variety of monitoring software that act as excellent exception detection and alerting tools, but in practice, few frameworks that provide application developers with structured, robust error correction tools for use *within* their software packages.

Native Java¹ Exception Handling

The lowest-level exception scenario that developers encounter in Java is the plain try/catch/finally block in Java, as in:

```
try {  
    doSomething();  
} catch (Exception anException) {
```

¹ Examples and code fragments are given in the Java programming language but the concepts presented can be applied to any object-oriented language.

```

        handleException();
    } finally {
        finishUp();
    }
}

```

For simple error handling, such as checking some variables or attempting to access an I/O device, this rudimentary system works fine. When an error occurs, there is usually no alternative but to handle the low level exception inline and return some status to the calling stack frame. However, in modern distributed systems, there are more advanced requirements for error handling. Robust applications are required to retry failed steps, notify other components of failures and produce logging statements, among others. It can be argued that a more advanced, generic, pluggable exception handling framework would be a valuable tool for building distributed applications. To this end, a declarative chain of responsibility pattern can act as the foundation for building such a system.

Chain of Responsibility (a.k.a. Chain of Command)

The GoF chain of responsibility pattern's intent is to “avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request” [Gamma]. Essentially, a message object is passed through a chain of objects implementing a common interface. The interface includes methods to chain the handler together; i.e., get- and setSuccessor methods:

```

public interface Handler {
    public void handle(Object message);
    public void setSuccessor(Handler successor);
    public Handler getSuccessor();
}

```

Implementations of this interface use the successor property to pass the message through the chain, as in:

```

public class HandlerImplementation {
    public void handle(Object message) {
        // local handling code here
        if (getSuccessor() != null) {
            successor.handle(message);
        }
    }
}

```

```
        }  
    }  
}
```

In the case of an exception handling framework, we wish to construct an object that encapsulates all of the information about the state of our application when an error occurred, as well as information about what to do for specific errors. For example, if we have implemented a Handler to retry database calls it's necessary to provide information as to where the secondary database instances are (see the "The Problem Interface," below).

ChainFactory

The chain of responsibility pattern does not really specify where the construction of the chain should occur; this, presumably, is left up to the application developer. Often it occurs in the head of the chain, but if the head is hard-coded into the program it immediately limits the flexibility of the software.

To circumvent this problem, a ChainFactory can be built. A ChainFactory produces the head of a particular pre-built chain that can be mapped to an identifier. For example, if a String catalogues ChainFactories, they could be requested from the factory, as in

```
ChainLink specificFactory =  
    ChainFactory.create("chainForSQLExceptions");
```

Depending on the type of error, a specific set of logic can be applied. This creates the ability to write context-sensitive error handling code, and in so doing, create an opportunity to write intelligent, self-correcting code.

Declarative Programming

Declarative programming refers to the practice of refactoring application logic out of the code and into a reference file. The Jakarta Struts framework is an excellent example of a declarative system: application execution path is determined by a configuration file that describes which pages to load upon specific user actions [Jakarta]. A declarative approach facilitates quick system maintenance and the ability to "hot swap" features of a deployed application.

Putting a declarative slant on chain of responsibility, it's possible to define the specific chains, catalogue them by key and have the system load these serialized values into a table upon command. Any number of formats could be used to store this information, an example in XML could be:

```

<?xml version='1.0', encoding='UTF-8'?>
<error-chains>
  <error-chain name="chainForSQLExceptions">
    <chain-link class="com.myapp.LoggingChainLink"/>
    <chain-link class="com.myapp.PageMeChainLink"/>
    <chain-link
      class="com.myapp.RetryDatabaseChainLink"/>
  </error-chain>
  <error-chain name="defaultChain">
    <chain-link
      class="com.myapp.ReportToCommandCenterChainLink"/>
  </error-chain>
</error-chains>

```

A load() method could be provided to the ChainFactory which reads and parses such a file, building a Map of information. Additionally, keys need not be strings, they can be any object.

The Problem Interface

An important part of using chain of responsibility is standardizing the interface of the message passed down the links. If a wrapper around the underlying exception, error or application-generated warning is applied; information can be shared among nodes. An example interface could be:

```

public interface Problem {

    public void setAttribute(String name, Object obj);

    public Object getAttribute[(String name)];

    public boolean hasAttribute(String name);

}

```

Implementations of Problem could be created for specific recurring situations, such as:

```

public class DatabaseAccessProblem implements Problem { ... }

public class ProblemInRequestProcessor implements Problem { ... }

public class SecurityProblem implements Problem { ... }

```

Each specific implementation can store information that allows for some logic to occur. Coupled with a declarative handler chain, a framework evolves that allows for robust,

extensible generic exception processing subsystems to be designed and quickly adapted to any project.

Self-Correcting Code

Consider a set of implementations of Problem that contain context-specific callback methods on their source, as defined in an interface such as:

```
public interface Retryable {  
  
    public boolean retry(); // returns true for success  
  
    public int getMaxRetryCount();  
  
    public void fail();  
  
    public List getProblems();  
  
    public Problem getLastProblem();  
  
    // etc.  
  
}
```

An Object that implements Retryable has methods that allow external objects to trigger specific events and query as to the results of those events. For example,

```
1 public class RetryableDAO2 implements Retryable {  
2  
3     public boolean makeCall() {  
4         try {  
5             // JDBC code  
6         } catch (java.sql.SQLException sqle) {  
7             Problem newProblem = new  
8                 DatabaseAccessProblem(this);  
9             addProblem(newProblem);  
10            return false;  
11        }  
12        return true;  
13    }  
14  
15    public boolean retry() {  
16        retryCount++;  
17        return makeCall();  
18    }  
19    // remaining methods skipped for brevity  
20 }
```

² DAO denotes a Data Access Object

Note how the problem is constructed passing the `this` keyword as a parameter to the constructor (line 8). As the `DatabaseAccessProblem` passes through the chain, one of the links can be given the opportunity to make callbacks to the `Retryable` methods of the `DAO`.

Advanced Problem Handling Knowledgebase Building

Once a problem handling chain has been devised, the more ambitious developer might build handlers that interface with a knowledgebase. The knowledgebase exposes methods like `registerProblem(Problem)`, `getSolutionsFor(Problems)`, etc. Applications, while being developed or run in a production environment, could interface with this knowledgebase and potentially save developers and control room operators time and energy.

In order for an interactive problem-solution conversation to take place, the `Solution` interface is defined:

```
public interface Solution {  
    public String getComments();  
    public void applySolution(Object problemContext);  
}
```

In the simplest case, a developer working on a Struts application receives an exception containing a pithy, useless message (sound familiar?). If a problem handling chain is executed as part of the application framework, a `KnowledgebaseHandler` could be defined and invoked as part of the chain. The `Problem` is serialized and sent to the `KnowledgebaseHandler`, which checks to see if this problem has occurred before, retrieves solutions, and returns the previous developer's comments in the `comments` property. If no such problem has occurred before, and the developer solves the issue, s/he can access a web application to enter a solution so other developers can benefit.

References

[Gamma] *Design Patterns - Elements of Reusable Object-Oriented Software*,
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
Addison Wesley. 1995.

[Jakarta] <http://jakarta.apache.org/struts>

Copyright © 2004 Stieglitech, LLC. All rights reserved. Portions of this document contain material copyrighted by other parties. No portion of this document may be reproduced for commercial purposes without express written consent of the copyright holder. Reproduction for personal or academic use is permitted. All source code is distributed under the GNU General Public License (GPL). See <http://www.gnu.org/gpl> for a copy of the GNU GPL.