Container-Managed Exception Handling for the Predictable Assembly of Component-Based Systems

Kevin Simons, Judith Stafford

Department of Computer Science
Tufts University
Medford, MA USA 02155
{ksimons,jas}@cs.tufts.edu

Abstract. Component containers provide a deployment environment for components in a component-based system. Containers supply a variety of services to the components that are deployed in them, such as persistence, enforcement of security policies and transaction management. Recently, containers have shown a large amount of potential for aiding in the predictable assembly of component-based systems. This paper describes an augmentation to the component container, called the Container-Managed Exception Handling (CMEH) Framework, which provides an effective means for deploying exception handling mini-components into a component-based system. This framework promotes a more effective handling of exceptional events, as well as a better separation of concerns, yielding a more predictable component assembly.

1 Introduction

The goal of this ongoing research is to develop a container-managed exception handling (CMEH) framework that facilitates the creation and deployment of modular exception handling mini-components in order to promote proper separation of concerns in COTS-based systems. Commercial component developers are generally not aware of the components with which their software will interact when used in an assembly, and are therefore written to be as reusable as possible. The Java™ 2 Enterprise Edition (J2EE™) framework allows for binary implementations of Enterprise JavaBean™ (EJB) components to be directly "wired" together in a deployment without any sort of "glue code". This can be accomplished via EJB metadata and Java reflection. Components do not need to be aware of each other, merely the interfaces that they provide and require. Directly connecting commercial-off-the-shelf (COTS) components provides a great many well known benefits, but also yields several problems with predicting the behavior of the system once it is assembled [4]. One such predictable assembly problem arises due to current exception-handling practices in component-based systems. Commercial components are designed with no knowledge of the components with which they interact, and therefore have no knowledge of the exceptional behavior of such components resulting in three possible exception-related situations: (1) Components making calls to other components will be catching generic exceptions with very little useful exception handling. (2) The result of a component operation may be considered exceptional by the system developer in the context of the current application, but the exceptional result is allowed by the calling component. (3) There may be exception results that could be easily handled by the developer without requiring exception propagation back to the calling component, which would most likely handle the exception poorly in the first place.

Component containers are a receptacle into which components are deployed, providing a set of services that support component execution [7]. With these services implemented in the container, the developer is allowed to concentrate on writing components in their domain of expertise. Containers, therefore, provide an excellent model for the separation of concerns. Furthermore, all calls made to components must be relayed through the containers, so containers provide an excellent means of crosscutting. Containers are currently showing a great deal of promise in aiding in the problem of predictable assembly [7].

The basis of our research is an augmentation the J2EE container known as the Container-Managed Exception Handling (CMEH) framework. The ability to handle exceptions outside of the components alleviates the problem of improper exception handling in commercial components. Giving the system developer the ability to deal with the exceptions in an application-specific context leads to more useful handling of exceptions and more predictable and robust system performance. Furthermore, abstracting the exception handling into the container helps alleviate the tangle that generally occurs in exception handling code within commercial components [3]. For this research, the EJB container used was the container provided as a part of the JBoss open-source application server[1]. Conceptually, this research focuses primarily on application-level exceptions. Memory usage, for example, is a system-level exception that can be handled within the framework. If the system developer needs to impose memory constraints not inherent in the Java virtual machine, these types of exception may be raised and handled by the CMEH framework.

2 The CMEH Framework

The CMEH framework allows system developers to quickly and easily deploy and manage exception handling components on a Java application server, allowing for more appropriate handling of component exceptions. The CMEH framework provides an event-driven model for handling exceptional behavior. By intercepting component method calls at a variety of points during method invocation and dispatching exception events at these points, the CMEH framework allows event handling code to correct the exceptional behavior of the system. There are three main events dispatched during a method invocation: `method-called`, `method-returned` and `method-`

1 www.jboss.org

`exception`. When a component method is called, the invocation is intercepted before it reaches the component and the `method-called` event is fired. Handlers listening for this event have the opportunity to perform any necessary processing, before the invocation is released and allowed to reach the component method. If the method returns properly, the container stops the propagation of the return value and fires the `method-returned` event, again allowing the appropriate handlers to perform their processing. If instead the component method throws an exception, the propagation of the exception is paused and the `method-exception` event is fired. There are two additional events, `test-component-state` and `recover-component-state` that are used to handle cases where exceptional behavior results in a component being left in an invalid state.

Handling of component method-exception event

When an application-level exception is thrown by a method of a component, system developers have an opportunity to handle the exception after it is caught by the CMEH framework. This can be done in a variety of ways. In the simplest case, the exception-handling code can simply re-throw the exception, and it will be propagated back to the calling component. This is the generic behavior of the EJB container before the introduction of this new exception mechanism.

One useful option when handling the exceptions thrown by a component is to convert the exception into a different subclass of the Java `Exception` class. This method allows for the exception to be translated into an exception that the calling component knows how to handle properly. Of course, knowing what exceptions a component can handle is not immediately obvious, and often must be discovered through use [4].

Another possible use of this mechanism is to stop the propagation of the exception altogether. Rather than propagating an exception back up the stack to the caller, the system developer may instead wish to return a value of the correct type to the caller. This will effectively allow the developer to return a default value to the calling component in the event of erroneous behavior, or to perform simple modifications to the return values in order to correct any possible errors that will occur when the return value reaches the calling component.

Handling of component method-called and method-returned events

The container-managed exception handling mechanism allows a system developer to check the arguments passed to a component method before the method has been executed. This provides the developer with several useful options. First, the developer can test the value of the arguments to ensure they are acceptable in the application and to the component being called. If they are, the method call continues as normal with the arguments being passed along to the called method. If the arguments are in any way out of range, the container exception handling code can raise an exception that will be propagated back to the calling component, effectively ending the method call. Again, the developer will be able to throw an exception that can be usefully handled be the calling component. Furthermore, the system developer can modify the values of the arguments, then allow the method call to proceed as normal, thus eliminating any erroneous behavior in the component receiving the method call.

Similar to the monitoring of arguments, this container model also provides the developer with the means to verify all return values returned by a component method. Once again, the return value can also be modified by the container exception code in order to ensure correct functioning of the system, or an exception can be propagated back to the caller.

Handling of test-component-state and recover-component-state events

During the execution of a component method, a component may be left in a state that is invalid in the context of the application. Often the application is notified of this by way of an exception. After a component method throws an exception, the exception either 1) propagates back to the caller of the method or 2) is translated or caught by a handler of the method-exception event. In the CMEH framework, after an exception is thrown by a component method, the framework fires a `test-component-state` event after the method invocation has concluded and before the control flow of the application progresses. By handling this event, the developer can write code to test the state of the component in question. If the developers event handling method determines that the component's state is invalid, a recover`-component-state` event is fired. By handling this event, the system developer has an opportunity to correct the state of the component before the application flow resumes. Exactly how to handle this event is well beyond the scope of this research, but the CMEH framework provides a means of handling invalid component states in a modular fashion.

3 Implementation Details

In the CMEH framework, handlers for the various events are created by implementing a set of simple interfaces. The following interface is representative of the entire set of interfaces that allows the system developer to handle each of the five exception event types:

```
public interface MethodExceptionEventHandler {
  public Object handleMethodExceptionEvent(
                                      MethodExceptionEvent event,
                                      Invocation methodInvocation)
                                      throws Exception;
}
```

In order to deploy their exception event handling code, the system developer must modify the XML deployment descriptor of the EJB whose methods they want to monitor. The system developer must add a new tag into the `<assembly-descriptor>` portion of the deployment descriptor, as follows:

```
<assembly-descriptor>
```

```
<exception-handler>
     <method>
            <ejb-name>TestEJB</ejb-name>
            <method-name>create</method-name>
     </method>
     <method-called-handler class="org.ks.eh.mcHandler1"/>
     <method-exception-handler class="org.ks.eh.meHandler1"/>
     <test-component-state-handler class="org.ks.eh.tcsHandler1"/>
</exception-hander>
…
```

The `<exception-handler>` tag is formatted much in the same way as `<method-permission>` and `<container-transaction>` tags. The `<method>` tag specifies which method is to be handled by the container-managed exception handler. Each of `*-handler` tags specify the fully-qualified Java class to use in handling that event. It is perfectly valid to specify the same event handler class for several different component methods, and it is also valid to specify several handlers to handle the same event for the same component method, allowing exception handling code to be further modularized.

The next sections detail the JBoss application server implementation of the CMEH framework. Also covered is the reliance of the framework on several J2EE services, as well as services specific to JBoss.

3.1 The Interceptor Stack

In the JBoss application server, services (such as transaction and security) are wrapped around a client's call via the interceptor stack. The interceptor stack is a chain of stateless components that implement the `Interceptor` interface. This interface has a single method, `invoke`, that is passed a wrapped method call. The task of a single interceptor in the stack is to receive the invocation from the previous interceptor, perform any necessary processing, and then either pass the invocation on to the next interceptor, or throw an exception, effectively canceling the client's method call.

The interceptor stack is contained within the component container. The final interceptor in the chain is the container interceptor, which makes the actual call to the EJB method itself. The return value of the component method is then passed back up the interceptor stack, where once again the interceptors have the opportunity to perform operation on the invocation, pass the invocation further up the stack, or throw an exception back to the client.

The CMEH framework adds a new interceptor to the chain that is responsible for intercepting method invocations at the appropriate times and dispatching the exception events.

3.2 The JMS-based exception event model

The exception event model in the container-managed exception handling framework is based on the Java Messaging Service (JMS). This service, which is provided as part of the JBoss J2EE application server, provides a means of dispatching and listening for asynchronous messages. In JMS, a topic is a form of channel that listeners wait for messages on. When the system developer deploys their exception event handlers, the framework automatically registers them to listen on the appropriate JMS topic. When an event is fired by the framework, a new JMS message is created and then dispatched to the correct topic. Event handlers, deployed to handle the type of event that is carried by the JMS message, will receive the event and a new thread is automatically created by JMS for handling the event in. This allows for the framework to support asynchronous handling of exceptional behavior, which will prove helpful if several handlers are deployed on the same event for the same component method and some of the handling of exceptional behavior can be done concurrently. If several synchronous exception event handlers are deployed to handle the same event on the same component methods, the handlers receive the exception event in the order they were specified in the deployment descriptor. Specifying whether or not a handler is to be used asynchronously is also specified in the XML deployment descriptor.

3.3 The ExceptionHandlerService MBean

The exception handler service, responsible for the deployment of event handlers and dispatching JMS messages, is implemented as a Managed Bean or MBean in the CMEH framework. Other MBeans in JBoss include services for transactions and security. When an EJB wishing to use CMEH (as specified in the deployment descriptor) is deployed into the component container, the `ExceptionHandlerService` MBean deploys the event and registers them with the appropriate JMS topic so that they can have exception events dispatched to them. If the system developer deploys a new version of the exception handlers when the system is up and running, the `ExceptionHandlerService`'s class loader dynamically replaces the exception event listener object so that the new version will receive the events. When the CMEH interceptor in the interceptor stack receives the invocation, it uses the Java Naming and Directory Interface (JNDI) to look up the `ExceptionHandlerService` and instructs the service to dispatch a JMS message containing the appropriate exception event.

Implementing the service as an MBean allows applications running in other JVMs to look up the service via JNDI and register their exception event handlers with the services. This feature allows for exception handling code on entirely different machines to be registered with service in order to handle exceptional behavior in a distributed and parallel fashion.

3.4 Exception event automation

Some exception event handling patterns are useful enough that they have been automated in the CMEH framework. For instance, translation of exceptions can be accomplished with the following addition to the deployment descriptor:

```
<method-exception
class="org.tufts.exceptionservices.ExceptionTranslator">
```

```
  <translate-exception from=”java.io.IOException” to=”MyException”/>
</method-exception>
```

By adding this to the XML deployment descriptor, the system developer does not have to write any code, and the appropriate exception event handlers are created and deployed automatically. Other automated patterns include automatically testing the integer and string attributes of EJBs for the `test-component-state` event and automatic component reloading for the `recover-component-state` event.

4 Performance Overhead

Adding these exception handling mini-components to a software systems will introduce some performance overhead. There will be exception checking code running on method calls that rarely produce any erroneous or exceptional behavior. Certain efficiency steps must be taken by the developers of the exception handlers in order to minimize the costs of the exception handling. Since this framework has not yet been deployed with a large-scale software system, empirical results have not been collected, however the added benefits of proper separation of concerns, ease of development and predictability should far outweigh any costs.

On the other hand, there is a certain amount of overhead added to the system by the framework itself, even if the exception handlers themselves are very simple. Empirical data was collected for a very simple software system with only two components. Various tests were run to determine the amount overhead added to the system with varying number of dummy exception handling components. All tests were run on a Linux machine, with the JBoss server running on the Sun Java SDK 1.4.1.

Table 1. Latency Statistics for the CMEH Framework

	0 Syn, 0 Asyn	1 Syn, 0 Asyn	0 Syn, 1 Asyn	1 Syn, 1 Asyn
Method call	31ms	74ms	68ms	89ms
Method return	84ms	111ms	104ms	127ms
Exception	86ms	111ms	108ms	124ms

Currently, both the synchronous and asynchronous exception handlers are using JMS to provide the exception events. As a result, synchronous handlers incur a larger performance overhead. However, in theory, the synchronous handlers could be based on a much simpler mechanism (such as Java Events), cutting the performance costs. The future work of this research will focus a great deal on minimizing the overhead of the framework itself.

5 Conclusions and Future Work

Since commercial component developers are generally unaware of the exceptional behavior of the components their code will interact with, current exception handling techniques are insufficient for dealing with component-based systems. They fail to handle most exceptions in a useful way and they don't allow for handling exception behavior in the context of the application. The CMEH framework provides a means for system developers to handle exceptional behavior in their systems in a simple, modular, well-organized fashion. It provides automation for deploying and maintaining event handlers that are used to handle exceptions in an application-specific way. This system was primarily developed for dealing with COTS components, however modularity, separation of concerns and reduction of code tangle make it useful when developing proprietary components as well.

Currently, the implementation strategy is focusing on both adding new features as well as minimizing the overhead imposed by the system. The synchronous event handlers will be implemented using simple Java events and will be removed from the JMS portion of the framework. Also, in order to make the asynchronous event handling more useful, features are being added to automatically deploy event handlers on to remote application servers in order to handle exceptional behavior in a truly parallel fashion. This should greatly increase the speed when performing processor-intensive exception recoveries, providing the machines have low network latency. New features for automatic detection and recovery from invalid component states will likely not be added to the framework, as it is outside the scope of this research.

References

1. Agha, G. and W. Kim. "Actors: A Unifying Model for Parallel and Distributed Computing." Journal of Systems Architecture, 45(15):1263-1277, 1999.
2. Bass, L. et al. "Volume I: Market Assessment of Component-based Software Engineering." Technical Report CMU/SEI-2001-TN-007, Software Engineering Institute, May 2000.
3. Lopes, C. et al. "Using AspectJTM for Programming the Detection and Handling of Exceptions." Proceedings of the ECOOP Exception Handling in Object Oriented Systems Workshop, June 2000.
4. Stafford, J. and K. Wallnau. "Predicting Feature Interactions in Component-Based Systems." Proceedings of the ECOOP Workshop on Feature Interaction of Composed Systems, June 2001.
5. Tarr, P., and H. Ossher. "Hyper/JTM: Multi-Dimensional Separation of Concerns for JavaTM." Proceedings of the 22nd International Conference on Software Engineering, June 2000.
6. Tarr, P et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the 21st International Conference on Software Engineering, May 1999.
7. Vecellio, G., W. Thomas and R. Sanders. "Containers for Predictable Behavior of Component-based Software." Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering, May 2002.