

# Container-Managed Exception Handling Framework

Kevin Simons and Judith Stafford

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

## 1 Introduction

Commercial component developers are generally not aware of the components with which their software will interact when used in an assembly. Therefore, components are written to be as generic and reusable as possible. The components are not aware of each other, only of the interfaces that they provide and require. 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. This approach to deployment is the basis of Component-Based Software Engineering (CBSE), but it 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. Since commercial components are designed with no knowledge of the components with which they interact, they 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 have emerged as an area with great promise in aiding the predictable assembly of commercial components. Containers are a receptacle into which components are deployed, providing a set of services that allow 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. System developers configure these services in the container at deploy-time, but the implementation of such services is left up to the container developers [2].

Augmenting the J2EE container with 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; the code that controls normal behavior is separate from the code that handles exceptional behavior [3]. For this research, the EJB container used was the container provided as a part of the JBoss open source project<sup>1</sup>. This container is freely available, and is representative of the commercial J2EE containers.

## **2 Container-Managed Exception Handling**

The container-managed exception handling (CMEH) framework facilitates the creation and deployment of modular exception handling components to promote proper separation of concerns in COTS-based systems. 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 than is currently possible. It provides a modular, event-driven model for handling exceptional behavior. The CMEH framework is based on intercepting a component method call and dispatching exception events at a variety of points during the method invocation at these points in order to let event handling code correct the exceptional behavior of the system.

### **2.1 Handling of component method-exception event**

When an exception is thrown by a method of a component, it is caught by the container-managed exception mechanism, giving system developer an opportunity to handle it. 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. 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. An exceptionally useful option when monitoring the exceptions thrown by a component is exception translation. The exception handling code catches the thrown exception, extracts any needed information from the exception, and then throws a different class of exception. This method allows for the exception to be translated into a subclass of exception that the calling component knows how to handle properly. Of course, knowing what exceptions a component can handle is not immediately obvious. There is somewhat of a lack of consensus in the way component interfaces should be specified [2], and the exceptions a component expects to receive from components that it requires is often absent from current interface

---

<sup>1</sup> [www.jboss.org](http://www.jboss.org)

specifications. Unfortunately, a great deal of information about the way a component functions must still be discovered through use [4].

## **2.2 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 is executed; providing 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. On the other hand, 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 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.

## **2.3 Handling of test-component-state and recover-component-state events**

If a component throws an exception, there is a possibility that the component will be left in an invalid state. By handling the `test-component-state` and `recover-component-state` events, the system developer can logically handle this situation while keeping this exception handling code separate from the business logic of the components. 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 both cases, the container-managed exception framework fires a `test-component-state` event after the method invocation has concluded and before the control flow of the application progresses. This event gives the developer an opportunity to test the state of the component that threw the exception. If the exception event handling code determines that the component is in an invalid state, a `recover-component-state` event is fired. By handling this event, the system developer has an opportunity correct the state of the component before the application flow resumes. Handling of this event generally consists of unloading and reloading the component into the container.

## **3 Container Implementation**

To handle the exception events in the CMEH framework, the system developer need only write Java classes that implement a simple set of interfaces and then deploy their libraries on the application server. To 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. Once the `ExceptionHandler` classes have been developed, some additions are needed to the `ejb-jar.xml` XML deployment descriptor. Each EJB has an XML deployment descriptor that tells the application server how the bean should be deployed into the container. The system developer must add a new tag into the `<assembly-descriptor>` portion of the deployment descriptor. 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.

### **3.1 The Interceptor Stack**

The CMEH framework is dependent on a feature in the JBoss container known as 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 task of an 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 raise an exception, effectively canceling the client's method call. The return value of the component method is then passed back up the interceptor stack, giving the interceptors the opportunity to perform operation on the invocation, pass the invocation further up the stack, or throw an exception back to the client. CMEH 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. 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, receive the event and a new thread is automatically created by JMS for handling the event. Allowing the framework to support asynchronous handling of exceptional behavior. This will prove helpful if several handlers are deployed on the same event for the same component method and some of the exceptional handling behavior can be performed concurrently. Allowable synchronicity 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 done with a simple addition to the deployment descriptor specifying which classes to translate. By adding a few lines to the XML deployment descriptor, the system developer does not need to write any code, and the appropriate exception event handlers are created 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 Costs

The event-based nature of CMEH, along with required method invocations pauses, indicates that there is a definite performance cost for using CMEH. Empirical results as to exactly how much slower applications will run has not yet been collected, but the increased ease of development, the added system predictability and the proper separation of concerns is expected to outweigh any costs.

## 5 Related Work

The use of containers as a means of crosscutting is directly related to work being done in the field of Aspect-oriented Programming (AOP). Much like the CMEH approach, AOP for exception handling stresses the detangling of exceptional code, as well as support for multiple configurations, incremental development and dynamic reconfiguration [3]. CMEH is also influenced by research in the field of multi-

dimensional separation of concerns [6]. The hyperslicing mechanism that is a part of the Hyper/J<sup>TM</sup> tool [5] is a large scale abstraction of the type of crosscutting mechanism used in this component container research. The Actor-based approach also influenced both the separation of concerns focus and the asynchronous model of this research [1]. To the best of our knowledge, the work most closely related to CMEH is Vecellio et al.'s [7] research in augmenting the JBoss container. Assertion capabilities (including watchdog timers and software firewalls) were added to the EJB containers in order to better predict the behavior of component-based systems.

## 6 Conclusions and Future Work

The CMEH framework provides a simple and highly automated system for allowing system developers to deploy modularized event handlers for dealing with exceptional behavior in COTS-based systems. This framework promotes a separation of concerns and more appropriate handling of exceptions and other exceptional behavior. While the framework was developed mainly for use with COTS components, it will also help to alleviate code-tangle in proprietary component-based systems. Our work is ongoing and new features are being added. Exception handling will be more easily parallelized by adding support for automatically deploying exception event handlers on remote Java application servers.

## 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.