TUFTS-CS Technical Report 2004-2

June 2004

Container-Managed Exception Handling for the Predictable Assembly of Component-Based Applications (Master

by

Kevin B. Simons
Dept. of Computer Science
Tufts University
Medford, Massachusetts 02155

ABSTRACT

Component-based technologies have shown great potential for increasing developer productivity and reducing time to market for software systems. However, current component technologies fail to meet the quality attribute demands of the software industry [4], such as reliability and security. System developers are unable to properly predict the behavior of a system assembled from commercial off-the-shelf (COTS) components, and the source code for such components is often not available for modification. Component containers have great potential to more accurately ensure certain quality attributes of COTS-based systems. Containers provide a set of services to the components that execute in them. For example, containers have been used to facilitate security policies and transaction management. This research suggests augmenting component containers in order to allow for exceptions to be handled independently of the components. This modification to the component framework yields a better separation of concerns and more robust assembly of commercial components. Before quality attributes of component assemblies can be determined, the execution of the system must be made more robust by properly handling exceptions.

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

A thesis submitted by

Kevin B. Simons

In partial fulfillment of the requirements

for the degree of

Master of Science

in

Computer Science

Tufts University

Date

May 2004

Adviser

Judith Stafford

Abstract

Component-based technologies have shown great potential for increasing developer productivity and reducing time to market for software systems. However, current component technologies fail to meet the quality attribute demands of the software industry [4], such as reliability and security. System developers are unable to properly predict the behavior of a system assembled from commercial off-the-shelf (COTS) components, and the source code for such components is often not available for modification. Component containers have great potential to more accurately ensure certain quality attributes of COTS-based systems. Containers provide a set of services to the components that execute in them. For example, containers have been used to facilitate security policies and transaction management. This research suggests augmenting component containers in order to allow for exceptions to be handled independently of the components. This modification to the component framework yields a better separation of concerns and more robust assembly of commercial components. Before quality attributes of component assemblies can be determined, the execution of the system must be made more robust by properly handling exceptions.

Acknowledgments

I would like to thank Dr. George T. Heineman at the Worcester Polytechnic Institute for his comments and guidance during the course of this research work. The work done on component containers at the MITRE Corporation by Gary Vecellio was very influential on this work. I'd also like to thank Alexander Ran, Robert Katta and Mitri Abou-Rizk at the Nokia Research Center, Boston for their help with this work and Nokia, Inc. for allowing me to use a project there as a test application for this thesis. Last but not least I would like to thank my adviser, Dr. Judith Stafford, for all of her guidance and support while I was conducting this research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis introduces a framework for handling exceptions properly in component-based applications. Specifically, exception handling services were added to the Java 2 Enterprise Edition (J2EE^{TM}) framework. These augmentations address the deficiencies present in current exception handling techniques, as well as the problems that arise from assembling systems from commercial components.

The thesis will cover in detail the Container-Managed Exception Handling (CMEH) Framework, which allows system developers to handle exceptions in a more modular, application-specific context. The quality of the framework is validated by a series of benchmarking examples, as well as an example application. The exceptional behavior of the application is corrected using the CMEH framework, and the corrected version of the application is compared to the same application corrected with simple glue code wrappers.

1.1 Exception Handling

An exception is an event that occurs during the execution of an application due to anticipated erroneous behavior. This event interrupts the normal flow of the application's instructions and automatically transfers the flow of the program to special exception handling code. Numerous conditions during application execution can cause the raising of an exception; a few examples include out-of-memory errors, security policy violations, divide-by-zero-errors and general misuse of an object-oriented class Application Programming Interface (API). When such a fault occurs, an exception is dispatched (or "thrown") and control is transferred to a block of code set aside to handle (or "catch") the exception. In this case, an error is a mistake in the logic of a program, and a fault is a manifestation of that error.

Exception handling concepts have been around nearly as long as modern programming languages, with their introduction being sometime in the mid-seventies with the advent of the PL/1 and Ada programming languages. Similar constructs were later introduced in ANSI C, but not until C++, Ada 95 and Java did the exception handling constructs become ubiquitous in major applications. Today, it's nearly impossible to write a large scale application without including a significant amount of exception handling code due to the increased complexity of modern software systems.

There are two main classes of exceptions in modern programming languages: system exceptions and application exceptions. System exceptions are thrown by the system, generally due to resource issues, such as a database connection error. These exceptions are, for the most part, non-recoverable. While it may be possible to automatically reconnect to the database or bring up a backup server, it is very difficult to know what state the application is

left in as a result of the exceptional behavior. Application exceptions occur due to some violation of business-logic rules in the application itself. For example, an exception may occur when a developer attempts to set the size of a window on the screen to a negative value.

Most modern object-oriented programming languages (such as Java, C++ and Microsofts C# .NET) provide exception handling in the form of a `try` and `catch` block [1]. A `try` block is a construct that wraps around a piece of code that may raise an exception. If any instructions within a `try` block throw an exception, the flow of control is automatically transferred to the `catch` block that corresponds with the current `try` block.

```
1 try {
2         java.io.FileReader reader =
3                 new java.io.FileReader("test.txt");
4         String encoding = reader.getEncoding();
5         // code clipped
6 } catch (java.io.IOException ex) {
7         ex.printStackTrace();
8 }
```

Figure 1.1: A simple example of Exception handling in Java

In Figure 1.1, the creation of a new `FileReader` object may throw an exception if the specified file does not actually exist. If the constructor does in fact throw an exception, control is automatically transferred to the corresponding `catch` block without executing any subsequent line. In general, control is transferred to the inner-most `catch` block that can handle the appropriate type of exception. In this case, the `catch` block simply dumps the

[1] Since the focus of this thesis is on Java, Java will be used for all examples of exception handling constructs, although the constructs are very similar in C++ or C#.

execution stack of the application. In Java, `catch` blocks specify a particular class of exception to be caught. It is very common for a single `try` block to have several corresponding `catch` blocks, each catching a different class of exception. This sort of construction leads to much more useful handling of exceptions, since often the course of action for handling one type of exception will be different from other types. All exceptions in Java inherit from the class `java.lang.Exception`. Using polymorphism, it is possible to simple catch exceptions of class `Exception`. While this would allow all `try` blocks to only have a single corresponding `catch` block, it would not lead to useful handling of exception, since very little information is available to developers using the `Exception` base class. In Java, system exceptions all inherit from the class `java.lang.RuntimeException`. Exceptions that are subclasses of this class are considered unchecked exceptions and are generally only caught by low-level code, generally in the Java Virtual Machine (JVM) itself. Some Java resources also specify a third type of exception known as a JVM exception, which generally includes the `OutOfMemoryException`, however any attempt to detect or recover from these exceptions is outside the scope of this thesis. Application exceptions are any exceptions that do not inherit from `RuntimeException` in Java.

Now that the `try-catch` mechanism is well understood, it's important to understand where exceptions arise from in the first place. In Java, exception events are fired using the `throw` construct. Any method that may throw an exception must be identified using the `throws` modifier in the method prototype (see Figure 1.2).

Method prototypes may either specify that they throw the generic `Exception` class, or they may list several specific exception in a comma-separated fashion. The second is more descriptive to developers calling

```
1 public double divide(int num, int den) throws Exception {
2         if(den == 0) {
3                 throw new DivideByZeroException();
4         }
5         return num / den;
6 }
```

Figure 1.2: Illustrating the throws clause in Java exception handling

the method, but both constructions work. However, if only the generic `Exception` class is specified in the method signature, the code responsible for catching exceptions from this method may only catch the generic exception. Compilation will fail if specific classes of exceptions are specified in `catch` blocks if the same exception classes are not specified in the `throws` portion of the method signature. Only application exceptions are specified in a `throws` clause, since system exceptions are unchecked. An attempt to specify an unchecked exception in the `throws` clause of a method signature will result in compiler errors. When a method is invoked, the context of that invocation is placed onto the virtual machine's call stack. If that method calls another method, another context is added to the call stack. When a throw instruction is executed, control moves back up the call stack to the innermost `try` block that has a corresponding `catch` block prepared to handle the class or a superclass of the exception. It is not required to have a throw clause in a method in order to have the `throws` modifier in the prototype. If a method is calling another method that may throw an exception, it is perfectly acceptable for the outer method to also throw the same type of exception and omit the `try-catch` clause all together. However, if a method calls another method that may throw an exception, the calling method is required to either provide a `catch` block for that type of exception or to specify that it may

also re-throw the exception. All of these constructs are enforced by the Java compiler at compile time.

1.2 Component-Based Software Engineering

1.2.1 An Introduction to Component-Based Development

The field of Component-Based Software Engineering or CBSE is concerned with assembly of software systems from pre-existing software components. There is certainly no unanimous agreement as to what constitutes a software component, but for the purpose of this thesis, components will be defined as "binary units of independent production, acquisition and deployment that interact to form a functional software system" [24]. Simply stated, a component is a deployable, compiled unit of some functionality. Components are a unit of software reuse. Reusable components are assembled by system developers, rather than developing a large software application from scratch, line by line. A component is specified by the *interface* it provides; that is to say components are black boxes. An interface specifies the methods that may be invoked on a component. Developers need only be concerned with the functionality that components provide, not the implementation of that functionality.

There are a number of significant advantages to leveraging CBSE when developing software systems. First, in order to create components, software requirements must be well modularized. This means that requirements engineers and architects will have to break the system down into a modular, componentized architecture, rather than the large, monolithic style archi-

tectures of the past. The benefits to having a modularized architecture are well-known and will not be discussed in depth here, but briefly, modularized architectures make software systems 1) more adaptable, 2) more scalable and 3) more maintainable.

Secondly, developing reusable software components has the ability to dramatically cut the cost of software development. While it does in fact incur a cost of up to 250% to initially develop software components so that they can be reused [24](meaning adding certain generality to make them application non-specific, as well as proper documentation and packaging), there is a clear benefit to be able to reuse the proprietary software. Often a company will require similar functionality in several system and being able to develop a component only once and then reuse it several times is of great benefit when compared to attempting to extract a portion of code from an existing system and adapting it to fit the needs of a new system. This type of extraction generally requires the application of a great deal of code wrappers and "glue code" and rarely produces a stable and satisfactory system in practice. Furthermore, even successful systems created using this technique will be far less adaptable and maintainable than component-based systems.

Perhaps the most compelling argument for the use of component-based development is the possibility of the establishing of a software component market [24]. Rather than developing proprietary components, a software development organization will be able to browse a selection of commercial-off-the-shelf (COTS) components and purchase a component that fits their needs. With such a market in place, there will be several benefits for system developers. First, the cost of development will be dramatically cut, since the cost of purchasing a component will often be dramatically less than paying a developer to code similar functionality. Second, development time will

obviously be cut since purchasing a component is less time intensive than developing the functionality in house. Finally, the quality of software components should dramatically increase both because of large scale use as well as the competition within the market environment. The aforementioned reasons make CBSE an attractive means of creating software and ensure (more or less) that CBSE is not just another software fad and that it has actual applications to future software development.

Component Models

A *component model* essentially defines what it means to be a component. It is a contract between component developers and the system that hosts the components, describing how the components should be developed and deployed. In most cases, this involves specifying a set of interfaces that a component must implement in order to be deployed into a component framework. There are several very popular component models in existence today, such as CORBA, Microsofts COM and .NET and Enterprise Java Beans, part of the Java 2 Enterprise Edition (J2EE) framework. Again, since this thesis focuses on Java, EJB will serve as an example of component models.

In the EJB model, a component must implement several life-cycle management methods before it can be deployed into the component framework. The EJB component is a server-side component model (unlike classic JavaBeans), specifying a contract for distributed components. There are essentially two key interfaces in the EJB model: the home and remote interfaces. The home interface specifies life-cycle management operations and is primarily made up of a `create()` method and a `findByPrimaryKey()`

method[2]. The remote interface specifies the set of business-logic methods that the component provides to client components. In EJB 2.0, there are two new interfaces called the local-home and local-remote interfaces which present optimized implementations of the home and remote interfaces for components residing on the same physical machine.

The EJB 2.0 model provides interfaces for the creation of three types of components, known as Session, Entity and Message-driven Beans; These three types provide component developers with support for stateless, stateful, and asynchronous components, respectively.

Component Frameworks

By meeting the necessary standards of the component model contract, a developer compiles and packages a binary unit that is ready to be deployed into a component framework. A *component framework* is a component deployment environment, which provides a set of important deployment-time and runtime services to components. Examples of services include component life-cycle management, security policy enforcement, transaction management, and persistence. Most frameworks, including J2EE application servers, provide these services in a dynamic and transparent fashion to the running components. In the J2EE framework, these services are configurable by the system developer using a set of XML files, called the EJB "deployment descriptors", and can be used or not used depending on the particular needs of the system.

[2]While these are the main methods of the home interface (for stateful EJBs), developers are free to develop other create and find methods in their remote interface.

1.2.2 The Problem of Predictable Assembly

Unfortunately, most of the aforementioned benefits of component-based software engineering can only presently be realized through in-house proprietary components. This is because the black-box nature of commercial components makes it very difficult to predict their behavior when they are assembled into a large software system. It is nearly impossible to predict the extra-functional properties (or quality attributes) of COTS components that are a part of a larger system. In CBSE, this is known as the problem of *predictable assembly*. Often, component vendors will provide some empirical data relating to the performance or reliability of their components in the documentation of the component, but there are no assurances as to how this component will behave while interacting with other components. Not being able to predict these features make the use of COTS components extremely costly both in terms of money and time, since a software developer must essentially purchase a component first and then test its extra-functional properties in the context of the system. This may lead the developer to discard a component after its been purchased due to its failure to meet certain quality attributes [15]. Of course, most software vendors will not be willing to incur these additional costs. These points will be further illustrated later in this thesis (see Section 5) during the discussion of an example web application. This application uses several COTS components to process HTML from web sites, however, the black-box nature of the COTS components made the correctness are reliability of the assembly very difficult to predict. While current component frameworks attempt to alleviate this problem by enforcing certain quality attributes at runtime (e.g. security and latency via watchdog timers), these approaches are more a bandage on the wound rather than a solution. Until the problem of predictable assembly can be solved, and

quality attributes of components can be accurately determined and properly certified, the advantages of CBSE cannot be fully realized.

Since the implications of this problem are very clear, a great deal of the research in the field of CBSE presently focuses on predictable assembly. Some of this work involves augmenting the components themselves in order to make the assembly more predictable. One such example involves analyzing the component architecture for potential areas of deadlock (a situation where all processes are waiting and thus none can proceed) and livelock (a situation where a change in one process cause a reaction in another, which in turn causes a reaction in the first, infinitely) [10]. Once these danger zones are located, new components are automatically generated and inserted into the system. These new components are capable of breaking the lock situations and can gracefully restore or restart the system.

A change to component models to allow for runtime assertion checking is the focus of ongoing research at Worcester Polytechnic Institute [9]. These assertions do not address the problem of determining quality attributes at design time, but they do allow properties of components to be asserted in a running system. In this proposed change, "hooks" are inserted into the components in the "before-phase" (before a component performs any steps toward fulfilling a request) and the "after-phase" (after the component has completed all execution steps for a request). These hooks allow contracts to be compiled into the code that enforce certain property requirements. These contracts aid in building trust between component users and component vendors, since guaranteed quality attributes can now be closely monitored. A Runtime Interface Specification Checker (RISC) compiler has been developed to automatically compile specified contracts into the code [9]. It also allows vendors to create two versions of their components; one with the aforemen-

tioned hooks and one without any additional overhead. Research is currently being done in applying these proposed model change to actual use cases.

Work being done at Microsoft Research focuses on the specification aspect of the problem of predictable assembly [3]. One of the main goals to be achieved in solving the problem of predictable assembly is to include promises about the quality attributes of components directly in to their specification. The researchers at Microsoft created a specification language called AsmL [3], which is an executable language that runs in the .NET runtime. Using this language, component developers can specify highly detailed interfaces for their components, leveraging features of AsmL, including non-determinism and atomic transactions. Since the AsmL language is executable, component developers specify their component interfaces with the AsmL languages and then enrich their specifications with attribute fields and method bodies. This is a major advantage over traditional programming language (C#, C++, Java) interface declarations. These method bodies ensure certain properties about the component method via pre- and post-conditions, declarative specifications and model programs. Rather than attempting to determine quality attributes about a component statically from these specifications, the AsmL specifications themselves are executed concurrently with the component code, and the results of the component operations are compared with the specification to ensure that the specified conditions have been met.

A great deal of research has been done in empirical studies of component performance [15, 19]. Before components can be certifiable, there must be a way to accurately describe and test their quality attributes. Certain attributes (e.g. latency) require empirical tests in order to determine the level at which a component meets them. These empirical studies are simple enough to do with freestanding components, but become difficult to predict in large

scale systems. A solution to this problem is being researched at the Software Engineering Institute. A general model for creating prediction-enabled component technologies (PECTs) is being created as part of the PACC (Predictable Assembly from Certifiable Components) project [12, 22]. The first generation of this model has been developed and it allows a component system to be developed in an environment that is capable of predicting latencies of the components in the system. This technology has been named ComTek-λ. Work on a second generation model that allows for the prediction of reliability is currently underway. This technology will also yield tremendous benefits to product lines. Since new products are constantly being created with new components, it is very important to maintain the same quality attributes in the products [12]. Also, if it is necessary to add a restriction on a quality attribute after the system has been created, a system created with PECT components should be able to handle this very well. A new method can be added to the components requiring this new quality attribute, and the system should be able to determine whether or not the current system meets its requirements.

Component performance is also the focus of the Component-Based Software Performance Engineering (CB-SPE) project [6]. This research, done at two universities in Italy, attempts to combine software performance engineering (a systematic, quantitative approach to construct software systems that meet performance objectives) and Real-Time UML (RT-UML, a set of UML domain profiles for schedulability, performance and time) into a single modeling language. By following this compositional approach, the researchers believe models that more accurately reflect the performance of component-based systems can be developed. This ongoing research is currently focusing on automating this modeling process.

A process for accurately making empirical measurements of component reliability has also been the focus of some recent research [15]. The CORE (COmponent REliability) measurement method, promotes a process for decomposing the specification into logical pieces about which it is possible to reason. The theory is that if it can be determined what role a component plays in a specific protocol (a sequence of service invocations between a set of components), then certain clues about a reliability measure to base the reliability of the assembly can be ascertained. The CORE method provides a seven step process for decomposing the specification into logical units, developing and applying a reliability test suite, and for analyzing the results to determine the reliability of the assembly.

Another proposed solution to the predictable assembly problem is to use a formal method to develop components [13]. Rather than attempting to specify the quality attributes of a component after it has been constructed, this approach proposes a development of a component based on quality attribute specifications. This *a priori* logical method uses formal transformations to develop components that are guaranteed to meet certain quality attribute goals.

However, these formal approaches tend not to scale well, so current research at NASA is focusing on using a technique as Design for Verification (D4V) in order to compose large systems from components that are independently verifiable [16]. This approach supports the component design process at two levels: first, it provides a set of domain specific design pattern styles. These pattern rules are sufficiently formal to allow for automated checking. The approach also provides an aspect-oriented implementation that supports CheckPoints (to ensure consistent program states), ControlPoints (to describe the process model of the system that are relevant for testing and

model checking), and ExtensionPoints (to denote potential extensions of the system).

Component Certification

Predictable assembly is further complicated by the idea of component certification [11, 13]. Currently, there is not a standard to label and document components. While a company can produce a component and claim that it meets certain quality attributes, their specifications of the component and they way in which the non-functional properties were determined is often poorly documented. The specification of the functional properties is often not the problem at all. There needs to be a universal standard in order to make the problem of component selection more manageable. If certifiable components are labeled, specified and documented with a universal standard, system developers will have a much easier time selecting components based on their non-functional requirements. This standard is currently the focus of a great deal of research. A component documentation scheme, proposed by Kallio and Neimel [11], consists of the components function and interface specifications, along with the constraints of the component, its performance measure, the security of the component, and a thorough description of the acceptance tests used in developing and testing the component. A standard similar to this would provide developers with a much more thorough basis on which to choose components.

Another proposed solution to the predictable assembly problem is third party certification of components [7, 23]. Whereas a developer may not be inclined to trust the component creators assessment of a components non-functional properties, they may trust the same component if it has been certified by a third party, based on UL 1998, 2nd ed., UL Standard for

Safety for Software in Programmable Components [7]. Third party certification is just a means to creating standardized documentation of components. However, many researchers feel that the task of establishing component certification and trust should be a distributed task and need not include third part certifiers [23].

1.2.3 An Overview of Component Containers

One of the most important pieces of the component framework, and the part responsible for providing a deployment environment for the components, is the component container. The container is a logical receptacle into which components are deployed. In the J2EE framework, it is the container that is responsible for providing services to the components that run in them. With all of the services implemented in the container, the components themselves need only contain business logic, and the service development can be left up to the container experts. Often there is a one to one ratio of containers to application servers and each container contains many components. However, this is not necessarily the case. It is possible to have many containers within a single application server (like in most J2EE application servers, where there is a different container for each component type), and it is also possible to only have a single component in a container. Often in J2EE, the terms container and application server are used interchangeably.

The J2EE container provides a variety of services to the components that are executing in it, including life-cycle management, data persistence, resource pooling, security policy application, transaction management and context management, among others. All interactions with component clients are performed by the component container. The container receives the request from a client, marshals the request to a component running in it, receives

the components response and marshals the response back to the client, as in Figure 1.3. This allows the container to provide services, such as security and transaction management. Component containers are virtually a large scale application of the adapter design pattern. The EJB container also performs all the life-cycle management by publishing its own remote interfaces to the clients, rather than allowing them to see the components actual interfaces. This allows the container to instantiate instances of EJBs only when they are needed in order to conserve resources. Furthermore, the container enforces reentrancy policies for non thread-safe components. Through Container-managed Persistence (CMP), the container automatically performs database persistence operations, allowing developers to write stateful components containing absolutely no Java Database Connection (JDBC) code simply by correctly configuring the aforementioned XML deployment descriptor.

Recently, the J2EE EJB container has shown a great deal of potential for aiding in the problem of predictable assembly [28]. Since the container is intercepting all of the calls to a components interface, it is an excellent means of applying a set of policies and contracts. Containers may prove to be highly influential in solving the problem of predictable assembly and they play a key role in this thesis research.

1.3 Current Enterprise Java Exception Handling Best Practices

The current focus in the field of EJB exception handling is recovering from component errors quickly with the goal of maintaining the end-user experience without dropping any user requests [18]. In order to improve efficiency,

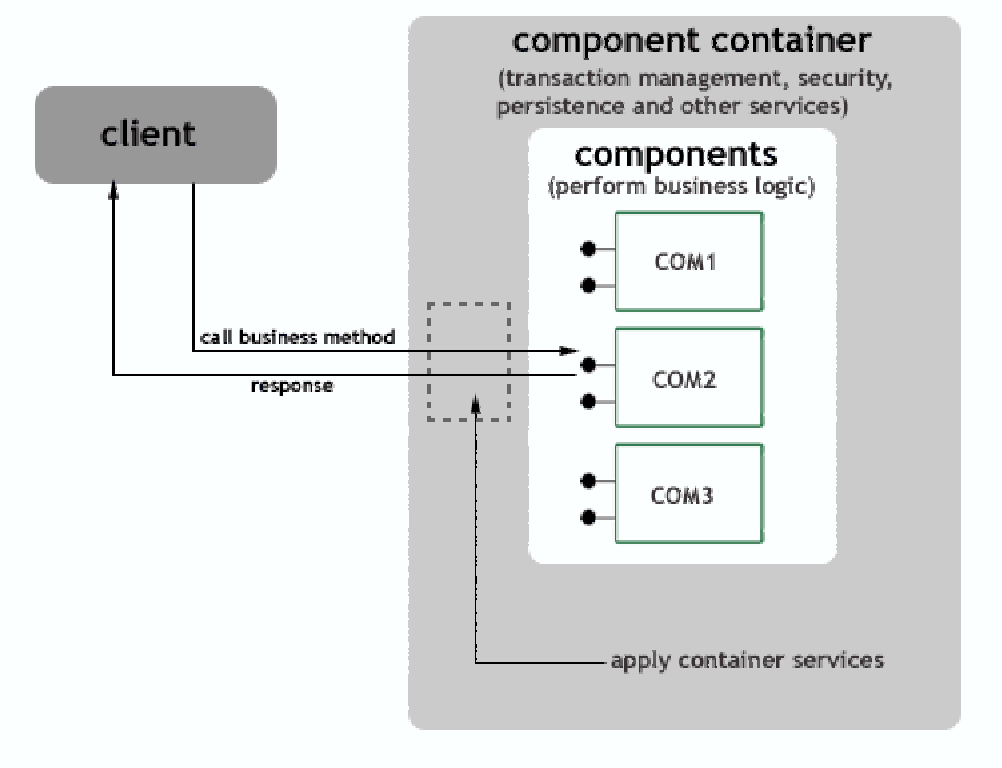

Figure 1.3: The Java 2 Enterprise Edition (J2EE) EJB Component Container

the old exception mainstay of simply printing out the execution stack trace is unacceptable in enterprise systems. Instead, appropriate logging and recovery technologies are leveraged.

When exceptions are currently caught in the EJB container, means are taken to try to ensure no data is lost and that the system remains in a working state. For example, when unchecked exceptions are caught, transactions are rolled back and the exception is wrapped up as some form of `javax.ejb.RemoteException` and is passed back to the client for proper handling.

Three basic techniques are used by developers of components who are calling methods of other components that may throw exceptions. Typically 1) the exception is rethrown with an error message, 2) the error is logged and the exception is rethrown or 3) the exception is wrapped in a different class of exception in order to preserve data across remote interfaces.

In general, clients rarely interact directly with Entity (stateful) EJBs. Instead, there is generally an intermediary Session (stateless) EJB that the client interacts with in order to prevent database corruption, etc. As a result, the Session bean can handle all of the exceptions from the Entity beans and produce some sort of error message or error recovery plan to the client.

While all of these techniques provide quite a good example for developing a system with proprietary components, they have several shortcomings. First of all, even simple applications suffer from a tremendous amount of code tangle when handling exceptions [14]. This is only complicated when remote components are involved, so handling all of the exceptions in a Session bean will be an overly complex and inelegant approach.

However, the real problem with current approaches is that they fail to meet the needs of systems dealing with COTS components. When a com-

ponent developer is creating a component, they do not know what other software their component will be interacting with. Therefore, they cannot possibly know what exceptions may be raised by the components that they rely on. In a true CBSE environment, commercial components may be directly "wired" together purely based on the interfaces they provide and require without any need for glue code [5]. When using EJBs, this can be done using EJB metadata. This means that the techniques mentioned above for dealing with proprietary components are completed irrelevant for this approach.

Furthermore, with COTS components, situations will arise when data being passed between components is allowed by the components, but may be considered exceptional in the context of the application by the system developer. An example of this is a method that takes a numeric argument, but there is a semantic mismatch of units between the calling and receiving components. Current CBSE exception handling constructions and techniques do not provide any means for dealing with this type of situation without using glue code. Wrapping components in this sort of glue code only serves to make the system less modular, less maintainable and less robust.

1.4 Summary

The field of component-based development offers a great deal of promise for reducing both time and cost of developing large-scale applications. Through component reuse, software development organizations will be able to purchase third-party components and assemble software systems rather than developing the systems entirely in-house.

However, currently there is a major problem facing the field of CBSE.

There is currently no means of accurately predicting the way in which a component will behave in the context of an application. The quality attributes of these components cannot be accurately determined. Without being to accurately predict these extra-functional properties, software organizations will not undergo the potentially expensive endeavor of developing software from pre-existing components.

While modern programming languages have well established exception handling mechanisms, these techniques for handling erroneous behavior do not map well to component-based systems. Particularly, when dealing with COTS components, the current EJB best practices are entirely insufficient. The aim of this research is to correct this problem by providing component-based system developers a highly modularized framework for handling exceptions in an application-specific context. This framework specifically addresses three aforementioned deficiencies in current exception handling best practices: exception type mismatch, semantic mismatches in method arguments and return values, and the detection and recovery from invalid component states.

Chapter 2

Technology Overview

2.1 Platform

Before the details of the framework developed as part of this research can be discussed in detail, it is first important to understand some of the major technologies that the framework depends on. A brief description of the platform used to run the framework will be described, followed by an in depth look at the J2EE technologies.

2.2 Platform

The research for this thesis was conducted on a machine running Redhat Linux 8 (Kernel 2.4.20-30.9). The Sun Java Runtime Environment (JRE) 1.4.1 was used to run all of the Java applications. All of the code was built using Apache Ant 1.5.4 using the Sun Standard Development Kit (SDK) 1.4.1.

2.3 The JBoss Java Application Server

The central piece of technology used in this research is the JBoss Java application server (version 3.2.2)[1]. JBoss is a freely available open-source project and is representative of the application servers currently available on the market. The JBoss server provides a fully open-source implementation of the J2EE framework and container, as well as the EJB 2.0 component model. The server is fully configurable via a set of XML configuration files and has an extensive set of available services and upgrades. The server also comes bundled with the Apache Tomcat HTTP web server and Servlet/JSP Engine (version 4.1.29)[2], which allows the JBoss server to host Java web components or "Servlets" as well as Java ServerPages (JSP). Integration of the Tomcat server allows system developers to create web applications easily that leverage both web and EJB components.

2.3.1 Hot-Deployment of Components

In the JBoss application server, EJBs can be dynamically deployed and undeployed when the server is already up and running. When a system developer has written Java code for an EJB and created the necessary XML deployment descriptor and configuration files, all of the files are bundled together into a Java Archive (`.JAR`) file. The XML files reside within the `META-INF` directory inside the archive file. In order to deploy an EJB to the application server, the system developer need only copy the file to the application servers deploy directory (`jboss-3.2.2/server/default/deploy` by default). When an archive is copied to this directory, the JBoss server au-

[1] www.jboss.org

[2] jakarta.apache.org/tomcat

tomatically unpacks the archive and examines the XML configuration files, using them to appropriately deploy the EJB. It uses the configuration files to determine if the server can provide all of the services the component requires at this time.

EJB components can likewise be un-deployed in a similarly simple and dynamic fashion. By deleting the archive from the deployment directory, the JBoss server automatically un-deploys the EJB component. If any other components are dependent on the EJB, these conflicts are found by the EJB server at this point and appropriate exceptions and errors are fired.

Web applications can be similarly hot deployed in the form of Web Archive (`.WAR`) files. The JBoss server automatically passes the appropriate classes and configuration files off to the integrated Tomcat web server.

2.3.2 The JBoss Interceptor Stack

The research conducted as part of this thesis is dependent on a framework in JBoss known as the interceptor stack. In the JBoss framework, services (such as transaction and security) are wrapped around a client's call via the interceptor stack. The interceptor stack is an ordered chain of stateless components that implement the `Interceptor` interface (see Figure 2.1). This interface has a single important method, `invoke`, that is passed a wrapped method call. From this object (of class `jboss.invocation.Invocation`), an interceptor can gather information about the current context of the method's execution. 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.

```
1  package org.jboss.ejb;
2
3  public interface Interceptor extends ContainerPlugin {
4          public void setNext(Interceptor interceptor);
5          public Interceptor getNext();
6          public Object invokeHome(Invocation mi)
7                                              throws Exception;
8          public Object invoke(Invocation mi)
9                                              throws Exception;
10 }
```

Figure 2.1: The JBoss Interceptor interface

By intercepting the call in the component container in this fashion, services can be easily applied. For example, when a method call is intercepted by the `org.jboss.proxy.SecurityInterceptor`, it is able to check the context of the invocation to ensure that the client making the method call has the proper credentials to make the invocation. If the credentials are acceptable, the interceptor will pass the `Invocation` on to the next interceptor in the chain, otherwise an exception will be raised and propagated back to the client. Other interceptors include interceptors for transaction services and for retrieving JDBC relationship data.

New interceptors can be added to the interceptor stack in two ways in JBoss. They can either by added to the default interceptor stack, or they

can be added to the interceptor stack for a particular EJB. The interceptor stack is configured with a simple XML configuration file, as in other parts of the JBoss application server (see Figure 2.2). Similarly to components in the application, the interceptor stack may be dynamically reconfigured during application execution.

```
<container-configurations>
        <container-configuration>
                <container-name>
                        Standard CMP 2.x EntityBean
                </container-name>
                <call-logging>false</call-logging>
                <invoker-proxy-binding-name>
                        entity-rmi-invoker
                </invoker-proxy-binding-name>
                <!-- XML omitted -->
                <container-interceptors>
                        <interceptor
        org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor
                        </interceptor>
                        <interceptor>
        org.jboss.ejb.plugins.LogInterceptor
                        </interceptor>
                        <!-- XML omitted -->
                </container-interceptors>
        </container-configuration>
</container-configurations>
```

Figure 2.2: An example XML configuration for the JBoss interceptor stack

2.3.3 Service MBeans

In JBoss, services on the server are implemented as Managed Beans or MBeans that plug into a framework called the Java Management Extensions (JMX). Each MBean implements a standard interface that allows them to be

instantiated by the server. There are four interfaces that an MBean may implement. They include Standard MBeans, Dynamic MBeans, Open MBeans and Model MBeans. The two most commonly used MBean types used in JBoss services are Standard MBeans and Dynamic MBeans. The interfaces of Standard MBeans (and thus the services they support) are statically defined, whereas the Dynamic MBeans publish the services they provide only when a component is ready to use them, adding a degree of flexibility. Examples of existing MBeans in JBoss include `JBossManagedConnectionPool` and `TransactionManagerService`.

New MBean services can be added to JBoss in a fairly straightforward manner. Base classes are provided to service developers that handle most of the boiler-plate that goes into creating a service. By extending these classes, the system developer need only implement a few simple methods for handling the life-cycle of a service (e.g. `startService()` and `stopService()`). After deploying the classes that make up a new MBean, a service developer just needs to configure the service with a simple XML configuration file, as illustrated in Figure 2.3.

```
1  <mbean code="myService" name=":service=MyService">
2          <depends>
3          jboss.mq.destination:service=Topic,name=aTopic
4          </depends>
5          <depends>
6                  jboss.mq:service=DestinationManager
7          </depends>
8          <depends>
9                  jboss.mq:service=InvocationLayer,type=JVM
10         </depends>
11 </mbean>
```

Figure 2.3: JBoss XML configuration for deploying new MBean services

After creating a new service, the Java code and XML configuration files are bundled into a Service Archive (`.SAR`) file. This file may then be copied to the servers deploy directory in order to deploy the service. The services can be dynamically deployed and un-deployed much in the same was as J2EE applications.

In order to leverage the services that are provided by MBeans, clients can look up the MBeans using the Java Naming and Directory Interface (JNDI), as in Figure 2.4. By using this directory service, components can look up services by name and then leverage the methods they provide.

```
1 InitialContext ctx = new InitialContext();
2 MyService aService = (MyService)ctx.lookup("java:/myService");
```

Figure 2.4: Using JNDI to look up MBean services in JBoss

2.3.4 Java Messaging Service

The JBoss application server provides a service for system developers known as the Java Messaging Service (JMS). This service gives the JBoss server the capabilities to support asynchronous messaging. JMS is an example of a Message-Oriented Middleware (MOM), and the JBoss implementation of JMS is referred to as JBossMQ. JBossMQ is a JMS 1.0.2b compliant implementation of JMS. This messaging service provides via a set of channels or "topics". Senders and receivers register themselves with the JMS topic and messages are sent asynchronously through the topic (see Figure 2.5). Any number of receivers may be registered with the topic and any registered client may send messages on the topic. The sender does not need to worry if there are one or more receivers waiting on the topic and the receivers need not be

concerned with the status of the sender. JMS topics may be configured to be one-to-one or one-to-many. They may also be declared "durable", meaning that if a receiver drops its connection at any point, it may reconnect and retrieve all messages that were sent during its absence. JMS messages can be simple text messages, or they can carry an object payload, as long as the object implements the `java.io.Serializable` interface.

```
1  TopicConnectionFactory tcf =
2  (TopicConnectionFactory)c.lookup("java:/ConnectionFactory");
3
4  TopicConnection topicConn = tcf.createTopicConnection();
5
6  Topic topic = (Topic) ctx.lookup("topic/myTopic");
7
8  TopicSession topicSession =
9  topicConn.createTopicSession(false,
10 TopicSession.AUTO_ACKNOWLEDGE);
11 topicConn.start();
12
13 TopicSubscriber subscriber =
14         topicSession.createSubscriber(topic);
15
16 subscriber.setMessageListener(new MyMessageListener());
```

Figure 2.5: Example of setting up a publisher and subscriber in Java Messaging Service (JMS)

JMS was primarily introduced to J2EE in order to support a new addition to the EJB model known as Message-Driven Beans. These beans, introduced in EJB 2.0, are interacted with via asynchronous messages provided by JMS. MDBs are full-fledged EJBs (meaning the component container provides them with all the typical services), but they don't publish a remote interface. Instead, they are only reachable via JMS messages. Unlike typical JMS receivers, MDBs can receive and process JMS messages concurrently.

There are several default topics that are automatically created at startup time by the JBoss server (e.g. securedTopic), and new topics may be added by the server administrator, as illustrated in Figure 2.6. The default topics are mainly leveraged in implementing Message-Driven EJBs.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
name="jboss.mq.destination:service=Topic,name=exceptionTopic"
        <depends
        optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
        </depends>
        <depends optional-attribute-name="SecurityManager">
        jboss.mq:service=SecurityManager
        </depends>
</mbean>
```

Figure 2.6: JBoss XML configuration for creating new Java Messaging Service(JMS) Topics

2.4 XDoclet Code-Generation Engine

When developing EJB components, there is a tremendous amount of boilerplate code and XML to write. There is generally a single file that contains the bulk of the implementation for the EJB (the class that implements `javax.ejb.EntityBean` or `javax.ejb.SessionBean`), and several files for the remote and home interfaces (possibly local home and local remote interfaces) and an XML deployment descriptor (as well as possibly JBoss-specific XML configuration files). Most of the XML, as well as the home and remote interfaces, can be directly discerned from the main implementation class with the help of some attribute-oriented programming. Attribute-oriented pro-

gramming is the idea that significance can be added to the code by adding meta-data. In the case of XDoclet[3], the meta-data that is added comes in the form of Javadoc tags. By adding a series of Javadoc tags (illustrated in Figure 2.7) and then running the code through the XDoclet preprocessor, all of the boiler plate code and XML configuration files are automatically generated. The XDoclet processor is a complete rewrite of the Javadoc parser, and has been highly optimized. It provides mechanisms for creating XML from specified schema, as well as Java code generation.

The XDoclet code generation engine was used in this research both for the creation of example EJB application and was extended to incorporate the new features of the research itself.

[3]xdoclet.sourceforge.net

```
/**
 *
 * @ejb.bean
 *      class="MyBean"
 *      cmp-vresions="2.x"
 *      jndi-name="MyBeanHomeRemote"
 *      name="MyBeanEJB"
 *      primkey-field="id"
 *      reentrant="false"
 *      schema="MyBean"
 *      type="CMP"
 *
 * @ejb.home
 *      remote-class="MyBeanHomeRemote"
 *
 * @ejb.interface
 *      remote-class="MyBeanRemote"
 *
 * ... code snipped
 *
 */
public abstract class MyBean
                        implements javax.ejb.EntityBean {

        /**
         * @ejb.interface-method
         */
        public abstract void method1();

        // code snipped
}
```

Figure 2.7: A simple example of using XDoclet to generate XML deployment descriptors

Chapter 3

Container-Managed Exception Handling

3.1 Problem Space

As stated in Section 1.3, there are currently a number of deficiencies with current exception handling practices in component-based development, particularly in developing with COTS components. Since COTS component developers are not aware of what components their software will be plugged together with, they cannot predict the exceptional behavior of those components. This generally leads to a lack of useful exception handling in component method calls. A typical scenario in an EJB component is illustrated in Figure 3.1.

These exception handling constructs are handling only generic exceptions, not exceptions specific to the component being called. This generally means that the exception handling code in the catch construct will mainly just print error messages or stack traces, rather than handling the exceptions in

```
1 try {
2         int i = component1.method1();
3 } catch (javax.rmi.RemoteException remoteException) {
4         // exception handling code
5 } catch(Exception exception) {
6         // exception handling code
7 }
```

Figure 3.1: Current exception handling techniques in Enterprise JavaBeans

a meaningful way.

Furthermore, if the calling component does try to handle a wider variety of exceptions in a more meaningful way, the result is generally a large amount of code tangle [14]. There will be a significant number of catch clauses, each with very complicated code, resulting in a code base that is both difficult to read and to reason about.

Another important drawback with current exception handling structures is the inability to handle exceptions in an application specific way and also to handle behavior that is considered exceptional in the context of the application that may not actually raise an exception. Because the exception handling code is hard-coded into the calling COTS component, it cannot be reconfigured on an application by application basis. Furthermore, legal values returned by a called component may be considered exceptional by the calling component even if the called component doesn't raise an exception. Likewise, arguments passed to a called component may be considered exceptional even if the called component method will accept the values. This is a subset of the partial matching problem in CBSE, in which two components are nearly compatible, but there is a semantic misunderstanding in method arguments and return values [29].

3.2 The CMEH Framework

The Container-Managed Exception Handling (CMEH) Framework was created as a solution to the aforementioned problems. The purpose of this framework is to provide component-based system developers with proper support for appropriately handling exceptions in a simple, highly modularized way. By using this framework, system developers should be able to quickly and easily develop exception handling code that deals with exceptions in their system in an application-specific context. This exception handling code will be logically modularized, as well as easily deployable and maintainable.

At the heart of the CMEH framework is the exception event model (see Figure 3.2). This model allows the developer to deploy exception handling code in the listener pattern, where their code is listening for specific exception events. These events are dispatched at a variety of times during the execution of a component method, giving the system developer several opportunities to correctly deal with the exceptional behavior of the system in an application-specific way. This model provides substantial cross-cutting support, yielding a proper separation of concerns in component-based systems.

The events in this model are divided into three distinct categories or phases: 1) exception prevention and analysis, 2) exception handling, 3) component state recovery. The first phase takes place during method calls and method returns. When a component method is called, the invocation of the method is interrupted and a `method-called` event is dispatched. By handling this event, system developers may properly examine the method invocation in order to check the method arguments being passed to the called component. If the arguments being passed to the component are not valid in the context of the application, the system developer's event handling code

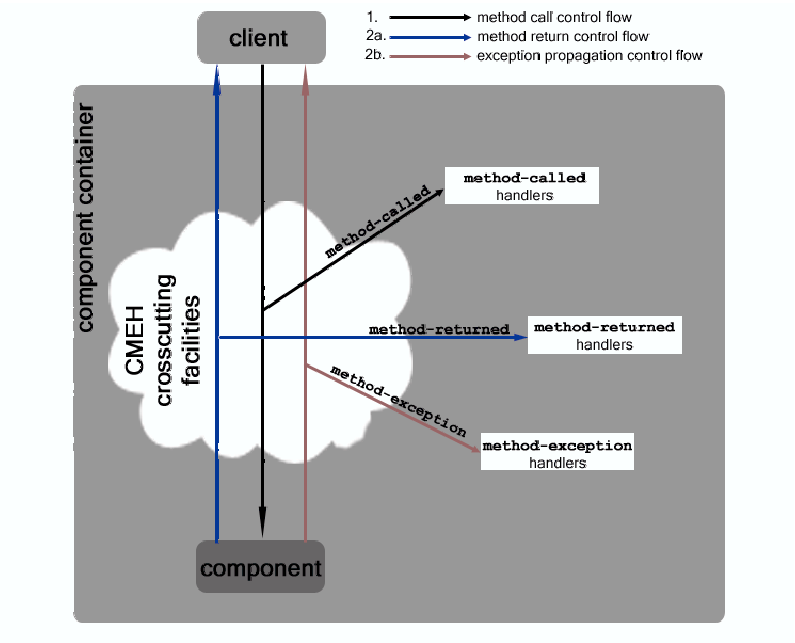

Figure 3.2: An abstract look at the CMEH Framework event model

can either correct the invalid arguments (if possible) or throw an exception that will propagate back to the calling component, effectively canceling the method invocation altogether. Since the exception event handling code is able to validate the arguments before they reach the called component, system developers can potentially drastically reduce the number of exceptions thrown by the called component method. Furthermore, they can also repair arguments that are invalid in the context of the application that may not ever raise an exception. Handling both of these situations will lead to a more predictable execution of the system, correcting any semantic partial-matching issues that may exist.

The second half of the exception prevention and analysis phase occurs when a component method returns from a call. Similar to a method call,

when a component method returns, the return value is stopped before it reaches the calling component and a `method-returned` event is fired. This event gives the exception-event handling code an opportunity to validate the return value before it reaches the calling component. Much like the `method-called` event, this event is useful for two reasons. First, it allows the system developer to correct any properties of the return value that may cause an exception when the return value reaches the calling component. Once the calling component receives this return value, it may perform further operations on the object, and the system developer would want to ensure that those operations are exception-free in an application-specific way. Furthermore, the properties of the return value may be exceptional in the context of the application, even though they would not necessarily raise an exception in the calling component. Therefore, it may be the responsibility for the CMEH event handling code to raise the exception. These semantic mismatches may also be corrected during this phase, yielding a more robust, predictable assembly of components.

The second phase of the CMEH event model is the exception handling phase. This is without a doubt the single most important phase in the event model, as it involves handling exceptions that could potentially render the system unusable. When a component method raises an exception, the propagation of that exception is intercepted before it reaches the calling component, and a `method-exception` event is raised. Exception event handling code listening for this event may perform several useful operations on the exception. One of the more useful operations is exception translation [28] or wrapping (see Section 1.3). As previously stated, the developers of the calling component did not know what component their component would be calling, they also were not able to predict what exceptions that component

would throw. Therefore it is very likely that the intercepted exception would not be handled in a useful way by the calling component. However, there undoubtedly exists a set of exceptions that the calling component does handle in a useful and robust way. Therefore, when handling the `method-exception` event, a possible option is to translate the exception that was thrown by the called component method into a type that the calling component knows how to handle properly. Of course, knowing what exceptions the calling component can handle properly is a difficult task without source code unless the component is extremely well document. However, a great deal can be discovered through use and exception classes native to the calling component's Application Programming Interface (API) are usually a good place to start [5].

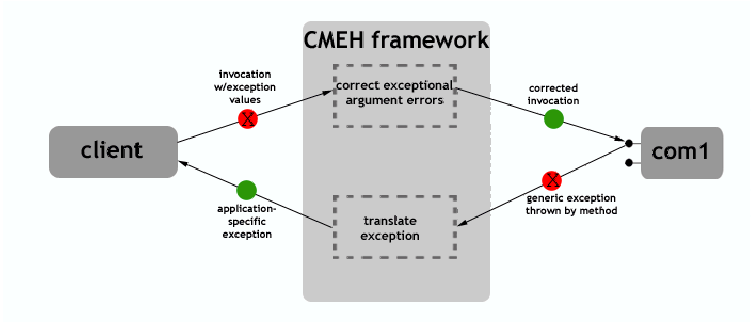

Figure 3.3: Translating exceptions using the CMEH Framework

Handling the `method-exception` event also provides an excellent means of instilling default values. Based on the type of exception thrown, component developers may intercept the exception and stop the propagation back to the calling component altogether. Instead, they may substitute a default return value to send back to the calling method. By using this technique, fewer exceptions are propagated back to components that would most likely

handle them incorrectly or generically in the first place. Returning simple default values of the correct type provides a much more appealing alternative.

The final phase of the CMEH event model deals with recovering components with invalid states. When a component method of a stateful component throws an exception, there is a reasonable probability that the called component is left in an invalid state. However, it is nearly impossible to code state recovery into the component itself, since the invalidity of the state must be determined in an application-specific way. Furthermore, a component moving into an invalid state will most likely cause a ripple-effect, where the components connected to the invalid component may have also been compromised. For that reason, the CMEH framework provides a means for detecting and recovering from invalid states. Whenever a component method throws an exception, the component may be left in an invalid state. Therefore, whenever this occurs, the CMEH framework fires a `test-component-state` event after the method execution has completed, but before control is returned to the calling component, regardless of whether an exception is propagated back the caller or a default value is instead substituted by a `method-exception` handler. Handling this event gives system developers an opportunity to test the validity of the called components state within the context of the application. If the `test-component-state` handler determines that the component is in fact in an invalid state, then the CMEH framework fires a `recover-component-state` event. Handling this event allows system developers to attempt to recover the invalid state of the called components, as well as any components that may have been affected by the exception. While this task is still very difficult due to dependencies among components, the framework does provide a means to handle the easily automated cases. Often handling this event will involve unloading and reloading a component

in order to recover its state. If an event handler cannot properly recover the state of a component, a new exception may be raised and returned to the caller, replacing any exception or return value that was currently being propagated back.

In order to handle the aforementioned events, system developers must create a set of event handling *mini-components*. These mini-components are highly modularized deployable pieces of code that allow the events to be handled within the context of the application. A mini-component is deployed on a specific event for a single component method. For example, a mini-component could be deployed for the `method-exception` event on `Method1` of `Component2`. The developer may deploy as many mini-components per method as is needed, allowing the handling of different exceptions, argument, and return values to be separated allowing for greater modularization and a better separation of concerns. On the other hand, a single class of event handling mini-component may be deployed on several different methods, across different components if necessary. Mini-components are deployable and highly maintainable, allowing them to adapt to the ever-changing needs of a component-based system. Just as most component frameworks provide a means to hot-swap components, the event handling mini-components may also be swapped out when the system is up and running, allowing the system developers to make updates to the exception handling code should the need arise.

3.3 Event Model Container Implementation

The CMEH framework is implemented as an augmentation of the J2EE container in the JBoss application server. In particular, the framework relies

heavily on the JBoss interceptor stack (Section 2.3.2) and service MBeans (Section 2.3.3), as well as the Java Messaging Service (JMS) (Section 2.3.4). While the interceptor stack and MBeans are specific to JBoss, a port to another Java application server would require a minimal effort, since most new servers support similar features. Furthermore, JBoss has become the *de facto* standard for freely available application servers.

Before examining the event model, it's important to understand the events themselves. The `ExceptionEvent` class provides a simple API (see Figure 3.4). In general, the ExceptionEvent provides the information that the CMEH framework needs to direct it to the appropriate event handlers and generally not information needed by developers creating their own CMEH event handlers. However, in certain cases (see Section 3.3.1) developers may need to query the `ExceptionEvent` to determine exactly what method invocation has triggered the `ExceptionEvent`.

```
1  public class ExceptionEvent
2          implements java.io.Serializable {
3          public ExceptionEvent(String eventType,
4                                String interfaceName,
5                                String methodName,
6                                Integer eventID);
7
8          public ExceptionEventContext getContext(
9                                        boolean async);
10
11         public String getEventType();
12         public String getInterfaceName();
13         public String getMethodName();
14         public Integer getEventID();
15 }
```

Figure 3.4: The public API for the ExceptionEvent

The `ExceptionEvent` also provides a means to acquire an `ExceptionEventContext` object associated with the event. The `ExceptionEventContext` object contains most of the information useful to the event handler mini-components (see Figure 3.5). These objects are related one-to-one with `ExceptionEvent`s and provide and API similar to that of a hash table. Some of the data provided by the context is mainly useful to the CMEH framework itself rather than the developers of event handlers, but it does provide a means to retrieve the `jboss.invocation.Invocation` object associated with the current method invocation. The `get()` and `put()` methods provide a means for several event handling mini-components deployed on the same component method to share data, as they share the exact same `ExceptionEventContext` object.

```
1  public abstract class ExceptionEventContext {
2          public ExceptionEventContext(HashMap hash);
3          public synchronized void put(Object key,
4                                       Object value);
5          public synchronized Object get(Object key);
6
7          public synchronized Semaphore getAsyncSemaphore();
8          public synchronized Invocation getInvocation();
9          public synchronized void setInvocation(
10                                      Invocation invocation);
11 }
```

Figure 3.5: The public API for the ExceptionEventContext

The exception event handler objects do not actually receive an instance of the `ExceptionEventContext` base class, but rather one of five subclasses that are specific to the type of CMEH event being handled (see Figure 3.6). Each of the subclasses are specific to a type of event and provide access to data relevant for that event. The `MethodCalledEventContext` ob-

ject, delivered to handlers deployed for `method-called` events, provides a `getArguments()` method that retrieves the method arguments for the current invocation. Likewise, the `MethodReturnedEventContext` provides a `getReturnValue()` to retrieve the object returned by the current invocation. Similarly, the `ExceptionEventContext` objects for the `method-exception`, `test-component-state` and `recover-component-state` all provide a `getException()` method to retrieve the exception thrown by the current method invocation.

Now that the events themselves have been discussed, it's important to understand how the events are delivered to the handlers. When the server is started up, the CMEH framework creates five new JMS topics for the purpose of receiving event dispatches (see Figure 3.7). Four other topics are created with the names `returnedTopic`, `exceptionTopic`, `testStateTopic`, `recoverStateTopic`. Each of the five topics is used as a communication channel for their related event types. While all events could easily be deployed on the same channel, using separate channels both cuts down on congestion in large systems and simplifies the code for receiving these events, since the event type will not need to be tested.

With the JMS Topics properly configured, the channels through which to dispatch the CMEH events are in place. At the heart of the framework is the `CMEHService` MBean. This service is responsible for the dispatching of exception events to the event handling mini-components, as well as the deployment and registering of the mini-components themselves. To this end, the MBean provides the public API illustrated in Figure 3.8.

The `createEventListener()` and `deleteEventListener()` methods are responsible for introducing new event handlers into the CMEH framework. These functions will be discussed in more detail in Section 3.3.1.

```
1  public class MethodCalledEventContext
2                          extends ExceptionEventContext {
3          public MethodCalledEventContext(HashMap hash);
4          public Object[] getArguments();
5  }
6
7  public class MethodReturnedEventContext
8                          extends ExceptionEventContext {
9          public MethodReturnedEventContext(HashMap hash);
10         public Object getReturnValue();
11 }
12
13 public class MethodExceptionEventContext
14                         extends ExceptionEventContext {
15         public MethodExceptionEventContext(HashMap hash);
16         public Exception getException();
17 }
18
19 public class TestStateEventContext
20                         extends ExceptionEventContext {
21         public TestStateEventContext(HashMap hash);
22         public Exception getException();
23 }
24
25 public class RecoverStateEventContext
26                         extends ExceptionEventContext {
27         public RecoverStateEventContext(HashMap hash);
28         public Exception getException();
29 }
```

Figure 3.6: The ExceptionEventContext subclasses

```
<mbean code="org.jboss.mq.server.jmx.Topic"
name="jboss.mq.destination:service=Topic,name=calledTopic">
        <depends
        optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
        </depends>
        <depends optional-attribute-name="SecurityManager"
        jboss.mq:service=SecurityManager
        </depends>
        <attribute name="SecurityConf">
        <security>
                <role name="guest" read="true"
                write="true"/>
                <role name="publisher" read="true"
                write="true" create="false"/>
                <role name="durpublisher" read="true"
                write="true" create="true"/>
        </security>
        </attribute>
</mbean>
```

Figure 3.7: One of the additional JMS Topics created as a part of the CMEH framework

```
1  public interface CMEHServiceMBean
2                  extends org.jboss.system.ServiceMBean {
3
4          public void createEventListener(
5                          String handlerClass,
6                          String eventName,
7                          String interfaceName,
8                          String methodName,
9                          boolean async);
10
11         public void deleteEventListener(
12                         String handlerClass,
13                         String eventName,
14                         String interfaceName,
15                         String methodName,
16                         boolean async);
17
18         public void dispatchMethodCalledEvent(
19                         Invocation mi)
20                         throws Exception;
21
22         public Object dispatchMethodExceptionEvent(
23                         Invocation mi,
24                         Object[] args)
25                         throws Exception;
26
27         public Object dispatchMethodReturnedEvent(
28                         Invocation mi,
29                         Object[] args)
30                         throws Exception;
31
32         public boolean dispatchTestComponentStateEvent(
33                         Invocation mi,
34                         Object[] args)
35                         throws Exception;
36
37         public void dispatchRecoverComponentStateEvent(
38                         Invocation mi,
39                         Object[] args)
40                         throws Exception;
41
42         public ExceptionEventContext getEventContextByID(
43                         Integer id,
44                         String eventType,
45                         boolean async)
46                 throws ExceptionEventUndefinedException,
47                 ExceptionEventIDUndefinedException;
48 }
```

Figure 3.8: The public API for the ExceptionHandlerService MBean

The MBean is responsible for dispatching all of the events to the mini-components using the JMS topics as communication channels. The creation and dispatching of events takes place in the `dispatch*Event` methods. When one of these methods is called, it is passed the `jboss.invocation.Invocation` object for the current method invocation. This object, along with a few other arguments provided to the method, is used to properly create the `ExceptionEvent` and associated `ExceptionEventContext` objects to be dispatched. After creating these events, the `ExceptionEvent` is published to the appropriate JMS topic and any handlers subscribed on that topic will receive the event.

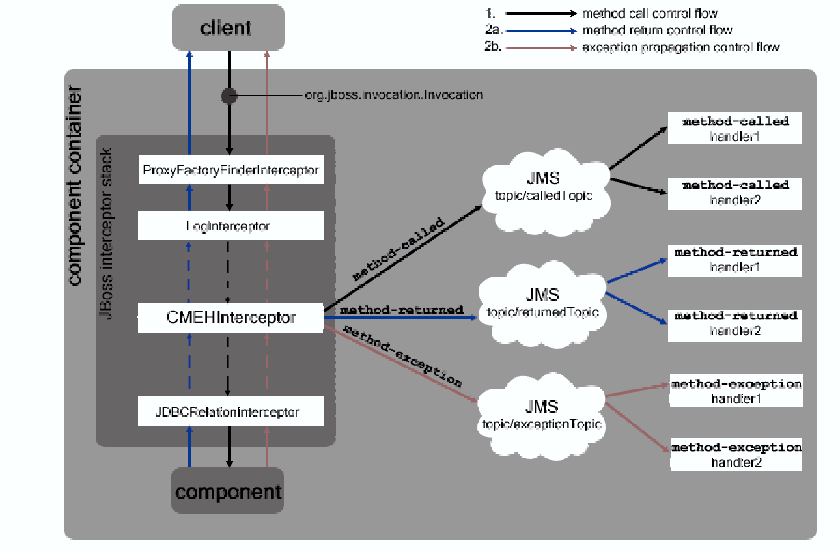

Figure 3.9: The JMS-based CMEH Framework event model

In order to provide the necessary cross-cutting and method invocation intercepting support, a new interceptor is added to the JBoss interceptor stack (see Figure 3.9). It's location in interceptor stack is significant, as it is located after all security policy and transaction enforcement. This eliminates

potential security concerns. This new interceptor is responsible for notifying the `CMEHService` MBean whenever a component method is being called, is returning, or is throwing an interception. At this point, the MBean can dispatch the appropriate event to the proper JMS topic. The interceptor's job is minimal, with the creation and dispatching of events actually taking place in the MBean.

The MBean and interceptor are packaged together in a SAR (Service ARchive) file. All of the XML configuration data for the MBean and the JMS Topics are contained in an XML file within the SAR archive called `jboss-service.xml`. Much like other deployment units in JBoss, the MBean service can be deployed and started simply by copying this archive to the deployment directory of the application server.

3.3.1 Event Handler Implementation and Deployment

Before examining the deployment and configuration of the CMEH exception event handlers, it is important to understand the details of the handlers themselves. The creation of the event handling mini-components is a very simple task. In order to create a new event handler, the system developer need only implement one of the interfaces in Figure 3.10 depending on which event they intend to handle. Each of these interfaces only require a single method that is called when it is required for the handler objects to handle CMEH events. The developer must also provide a default constructor for their handler classes. In general, each of the handler interfaces are quite similar, however they do have some differences. First of all, each of the `handle*` methods receive the same arguments with the exception of the `ExceptionEventContext` objects. Each of the handling methods receive the proper subclass of `ExceptionEventContext` relating to the type of CMEH

event to be handled.

The important detail is that each of the methods have different return values. For the `handleMethodCalled()` method of the `MethodCalledEventHandler` class, the expected return value is of the type `Object[]`. This array of objects corresponds to the arguments that will be used in the current method invocation. Therefore in order to modify the arguments, a `MethodCalledEventHandler` simply has to return a new array of objects, and that array will be used as the method arguments of the pending method invocation, rather than the original arguments. If any of the argument types to the called method are primitive types (i.e. `int`), they should be wrapped in their corresponding object types (`Integer`). The method can also throw an exception, effectively canceling the method invocation.

The `handleMethodReturnedEvent()` method of the `MethodReturnedEventHandler` class functions similarly, except the return value of this method is of type `Object`. This object represents the return value of the current method invocation, therefore to modify the return value, the handler object need only return a new object of the correct type.

The way in which the `MethodExceptionEventHandler` works is slightly different. Since this handler is used for handling `method-exception` events, the "typical" behavior of this handler is to throw an exception. In order to continue the propagation of the current exception, the `handleMethodExceptionEvent()` need only rethrow the `Exception` object acquired by the `getException()` method of the `MethodExceptionEventContext` argument. To translate an exception, the handler can simply throw an exception of a different type. However, if the handler instead returns an object of the correct type for the return value of the current method invocation, the propagation of the exception will be

canceled and the return value will instead be propagated back to the caller.

The `handlerTestStateEvent()` method of the `TestStateEventHandler` class return an object of type `Boolean`. This return value determines whether or not the component is in an invalid state. If the handler determines that the component involved in the current method invocation is in an invalid state, it can return a new `Boolean` with the value `true`. If the handler returns `true`, the CMEH framework will dispatch an event of type `recover-component-state` to the `handleRecoverStateEvent()` methods of the appropriate `RecoverComponentStateHandler` object. This method has no return value and is only used to modify the involved components.

After implementing one of these interfaces, the developer must specify how their new class is to be deployed via the XML deployment descriptor for the EJB whose events they wish to monitor (see Figure 3.11). Within the `<exception-handler>` tag, the `<method>` tag specifies the name of the component interface and component method to be monitored. The name of the interface is either the home or remote interface of the EJB. It then allows the developer to specify a series of CMEH event handler classes that should monitor specific events on the specific component methods via the `<method-called>`, `<method-returned>`, `<method-exception>`, `<test-component-state>` and `<recover-component-state>` XML tags. The XML deployment code should specify the fully qualified package name of the event handler classes. The `<exception-handler>` tags appear within the `<assembly-descriptor>` portion of the `ejb-jar.xml` deployment descriptor file. This section of the configuration file contains other application-wide configurations, such as security and transaction policies.

Typically, the classes for the CMEH event handlers are packaged together with the EJB they are associated with, though this is not a requirement.

```
1  public interface MethodCalledEventHandler
2                          extends ExceptionEventHandler {
3          public Object[] handleMethodCalledEvent(
4                                  Invocation invocation,
5                                  ExceptionEvent event,
6                                  MethodCalledEventContext c)
7                                  throws Exception;
8  }
9
10 public interface MethodReturnedEventHandler
11                         extends ExceptionEventHandler {
12         public Object handleMethodReturnedEvent(
13                                 Invocation invocation,
14                                 ExceptionEvent event,
15                                 MethodReturnedEventContext c)
16                                 throws Exception;
17 }
18
19 public interface MethodExceptionEventHandler
20                         extends ExceptionEventHandler {
21         public Object handleMethodExceptionEvent(
22                                 Invocation invocation,
23                                 ExceptionEvent event,
24                         MethodExceptionEventContext c)
25                                 throws Exception;
26 }
27
28 public interface TestStateEventHandler
29                         extends ExceptionEventHandler {
30         public Boolean handleTestStateEvent(
31                                 Invocation invocation,
32                                 ExceptionEvent event,
33                                 MethodTestStateContext c)
34                                 throws Exception;
35 }
36
37 public interface RecoverStateEventHandler
38                         extends ExceptionEventHandler {
39         public void handleRecoverStateEvent(
40                                 Invocation invocation,
41                                 ExceptionEvent event,
42                                 MethodRecoverStateContext c)
43                                 throws Exception;
44 }
```

Figure 3.10: The ExceptionEventHandler subclasses

```
<exception-handler>
        <method>
                <interface-name>
                com.tufts.cmeh.testsuite.TestSessionRemote
                </interface-name>
                <method-name>method1</method-name>
        </method>
        <method-called
        class="com.tufts.cmeh.testsuite.CalledHandler1"/>
        <method-return
        class="com.tufts.cmeh.testsuite.ReturnedHandler1"/>
</exception-handler>
```

Figure 3.11: CMEH XML configuration for deploying standard method-called, method-returned and method-exception events

When the archive file is copied to the application server's deployment directory, it is unpacked and appropriately deployed.

Invalid Component State Checking and Recovery

The deployment of the `test-component-state` and `recover-component-state` events is very similar to the other three events, but with a few subtle differences. First of all, handlers for these events are generally not deployed on a specific component method, just a component interface. This is because testing the validity of a component state and recovering from an invalid state will rarely be dependent on which component method rendered the state invalid. If it is important to know which method caused the exception, the method name can be retrieved from the event. Because of this slight change in deployment, the deployment descriptor for handlers of these two events generally is modified as illustrated in Figure **??**.

The special method-name value * is a way of specifying all of the methods

```
<exception-handler>
        <method>
                <interface-name>
                com.tufts.cmeh.testsuite.TestSessionRemote
                </interface-name>
                <method-name>*</method-name>
        </method>
        <test-component-state
        class="com.tufts.cmeh.testsuite.TestStateHandler1"/>
        <recover-component-state
class="com.tufts.cmeh.testsuite.RecoverStateHandler1"/>
</exception-handler>
```

Figure 3.12: CMEH XML configuration for deploying test-component-state and recover-component-state mini-components

in particular interface. While it is generally used with the `test-component-state` and `recover-component-state` events, it is valid to use it with other exception events as well. It is also valid to specify a specific method for the `test-component-state` and `recover-component-state`, but this is generally less useful.

3.3.2 CMEH Event Dispatching and Receiving Mechanism

When a new exception-handling mini-component is deployed into the system, it is automatically registered with the `CMEHService` MBean via the `createEventListener()` method. This method actually creates an instance of the `ExceptionEventListener` class, which is responsible for receiving CMEH events from the `CMEHService` MBean and marshaling the events to the appropriate handler. Each *listener* is associated with exactly one user

defined *handler* object.

When the `createEventListener()` method is called, a new `ExceptionEventListener` object is created and registered with the appropriate JMS `Topic`. Within this listener, a new instance of the appropriate event handler is created via the `java.lang.reflect` package. The system developer does not ever need to create or instantiate a `ExceptionEventListener` or an event handler object. This is always taken care of by the framework. The `createEventListener()` method is automatically called during deployment, however it is also valid for the developer to call the method programatically.

When an event is dispatched to a JMS `Topic`, each of the listeners registered on that topic receives the event. The properties of the event are tested to see if they handler associated with this listener should handle the event. If so, the listener passes the event off to the handler. If not, the event is ignored. It is this intermediary step of sending the event first to a listener and then to a handler that requires the differentiation between the `ExceptionEvent` and the `ExceptionEventContext` classes. JMS messages require that the object being sent implement the `java.io.Serializable` interface. Since the context provides hash table like functionality, there is no way to guarantee that the objects in the context are serializable. Therefore, the `ExceptionEvent` object serves as a simple handle, used to identify its associated context object. The event is serializable and can be sent to the listeners via JMS. The listener can then retrieve the context from the CMEH service and pass it off to its associated handler. All of this is transparent to the CMEH framework user, with both the `ExceptionEvent` and `ExceptionEventContext` objects being passed to the event handler objects.

Each of the event handlers registered for a particular event on a com-

ponent method execute in the order they are specified in the XML deployment descriptor. The CMEH framework accomplishes this by assigning each `ExceptionEventListener` object a unique ID. When an event is dispatched on a JMS channel, the CMEH MBean blocks until it receives notification that all of the listeners' event handlers have completed their processing. Each of the listener objects blocks until its turn comes (based on its ID), then marshals the event to its associated handler.

A timeout mechanism is required because of this blocking. If an event handler object contains an infinite loop, its associated listener will never notify the CMEH service that processing of the event is complete. For that reason, the service only waits a limited time before issuing a notification that the next listener in line should pass the exception event off to its associated event handler.

3.3.3 Asynchronous Support

Since the CMEH framework is built on top of an asynchronous communications layer (JMS), a logical extension of the framework is to provide support for asynchronous exception handling. By providing asynchronous handling of exception events, developers may concurrently execute several mini-components for exception handling tasks that are inherently parallelizable. The event model for the CMEH framework is slightly modified when asynchronous components are in use. Rather than dispatching the appropriate JMS event and then blocking until all of the event handling mini-components have returned, the CMEH framework simply dispatches the event and then continues the component invocation, allowing all asynchronous handlers listening for events on the component method to execute concurrently.

In order to make an exception event handler asynchronous, the developer

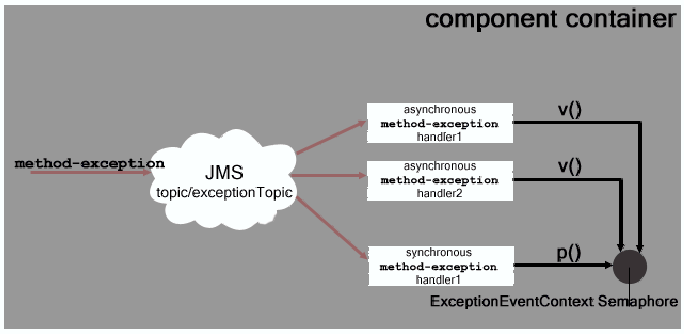

Figure 3.13: Asynchronous event handlers in the CMEH Framework

must modify the XML deployment descriptor configuration for the CMEH event handler components as illustrated in figure 3.14.

The exception handling mini-components themselves must also implement new asynchronous versions of the interfaces illustrated in Figure 3.15. The reason these new interfaces is needed is quite simple: since these new event handlers are asynchronous, they cannot return a value back to the caller as with the synchronous components. Asynchronous event handlers receive the same class of `ExceptionEventContext` as their synchronous counterparts. While these context objects do provide some synchronization, it's important for the developers of asynchronous event handlers to use caution when adding and changing values in the context, as it is possible for one asynchronous mini-component to clobber another's data in the context.

Using only asynchronous event handlers to handle exceptions on a component method does not provide any useful functionality. This is because the CMEH framework would dispatch the appropriate JMS events and then continue the invocation. By the time the asynchronous event handlers had finished their processing, the method invocation would be in an indeterminate state. Instead, a mix of asynchronous and synchronous event handlers

```
<exception-handler>
        <method>
                <interface-name>
                com.tufts.cmeh.testsuite.TestSessionRemote
                </interface-name>
                <method-name>*</method-name>
        </method>
        <test-component-state
        class="com.tufts.cmeh.testsuite.TestStateHandler1"/>
        <recover-component-state
        class="com.tufts.cmeh.testsuite.RecoverStateHandler1"
        async="true"/>
        <recover-component-state
        class="com.tufts.cmeh.testsuite.RecoverStateHandler2"
        async="true"/>
</exception-handler>
```

Figure 3.14: CMEH XML configuration for deploying asynchronous mini-components

should be used. This way, a single synchronous event handler can be included and it can block until all of the asynchronous components have completed, then return. By leveraging the event handlers in this way, the CMEH framework will block the invocation until the synchronous event handler returns, allowing the asynchronous event handlers to perform the required processing while the invocation is in a consistent state. The CMEH framework provides a simple synchronization system to facilitate this sort of construction (see Figure 3.13). The `ExceptionEventContext` hash contains a method named `getAsyncSemaphore()`, which return a semaphore used for event handler synchronization. This member is a semaphore whose value is automatically initialized to the number of asynchronous event handlers registered on the component method. Using this member, a synchronous handler can block trying to get the semaphore, while the asynchronous handlers each decrement

```
1  public interface AsyncMethodCalledEventHandler
2                          extends ExceptionEventHandler {
3          public void handleMethodCalledEvent(
4                                  Invocation invocation,
5                                  ExceptionEvent event,
6                                  MethodCalledEventContext c);
7  }
8
9  public interface AsyncMethodReturnedEventHandler
10                         extends ExceptionEventHandler {
11         public void handleMethodReturnedEvent(
12                                 Invocation invocation,
13                                 ExceptionEvent event,
14                         MethodReturnedEventContext c);
15 }
16
17 public interface AsyncMethodExceptionEventHandler
18                         extends ExceptionEventHandler {
19         public void handleMethodExceptionEvent(
20                                 Invocation invocation,
21                                 ExceptionEvent event,
22                         MethodExceptionEventContext c);
23 }
24
25 public interface AsyncTestStateEventHandler
26                         extends ExceptionEventHandler {
27         public void handleTestStateEvent(
28                                 Invocation invocation,
29                                 ExceptionEvent event,
30                                 MethodTestStateContext c);
31 }
32
33 public interface AsyncRecoverStateEventHandler
34                         extends ExceptionEventHandler {
35         public void handleRecoverStateEvent(
36                                 Invocation invocation,
37                                 ExceptionEvent event,
38                                 MethodRecoverStateContext c);
39 }
```

Figure 3.15: The Asynchronous ExceptionEventHandler subclasses

the semaphore one by one as they complete their processing. Developers are also free to manually synchronize the event handlers by any necessary means, using the `ExceptionEventContexts` `put()` and `get()` methods.

3.3.4 Event Handling Automation

Several event handling techniques are common and useful enough in the CMEH framework that they have been automated. This is done to save developers the time of coding simple event handlers. It also aids in maintainability since the developer does not actually have to write any code to deploy these handlers.

The most important of these automations is exception translation. By simply augmenting their component deployment descriptor, system developers can automatically add a `method-exception` event handler that translates an exception into a type that can be more usefully handled by the calling component (Figure 3.16).

```
1 <method-exception class="com.tufts.cmeh.Translator">
2         <translate from="java.io.IOException"
3                          to="MyException"/>
4 </method-exception>
```

Figure 3.16: CMEH XML configuration for automatically creating and deploying an exception translating method-exception handler

Using reflection, this automatically generated event handler attempts to recover as much state from the original exception as possible and transfer it to an instance of the translated exception. However, exception states rarely align well, so in general the state transference involves copying the exception

message.

Another important automation provided by the CMEH framework is the returning of default values in response to `method-exception` events. Presently, the framework only supports the returning of `String` and `Integer` values in response to exceptions. If the system developer wishes to deploy a default value generator for the method-exception event on an EJB, the code illustrated in Figure 3.17 is added to the XML deployment descriptor.

```
<method-exception
        class="com.tufts.cmeh.DefaultValueReturner">
        <exception class="IOException"
                        return="0"
                        type="Integer"/>
        <exception class="ParseException"
                        return="1"
                        type="Integer"/>
</method-exception>
```

Figure 3.17: CMEH XML configuration for automatically creating and deploying a default value returning method-exception handler

A default value may be returned for all exceptions for a particular method by specifying the `class="*"`.

3.4 Augmentations to XDoclet

As with other EJB deployment code, configuring and deploying CMEH mini-components can be quite cumbersome. By adding a series of new tags to the XDoclet code-generating engine, this problem has been alleviated somewhat. The method-level XDoclet tag illustrated in Figure 3.1 can be added to the

Javadoc comments of the component methods in the main EJB implementation class that will leverage the CMEH framework.

cmeh.exception-handler (0..*)		
Parameter	**Type**	**Description**
event-name	String	Which event should this handler listen for.
class-name	String	The CMEH event handler class.
async	Bool	Whether or not the handler is asynchronous.
generate-stub	Bool	Whether or not stub handler classes should be generated.

Table 3.1: New XDoclet tags for automatically generating CMEH XML configuration code

Adding this new Javadoc tag to the component methods that require CMEH and running the code through the XDoclet processor has several effects. First, The XML deployment descriptor of the EJB is automatically updated to include the necessary configurations for the CMEH mini-components. Furthermore, if the developer chooses, stub implementations of the event handler classes can automatically be generated, implementing the appropriate synchronous or asynchronous exception event handler interfaces.

Chapter 4

Benchmarking Tests

With the addition of any crosscutting facilities, particularly an event-driven framework, a certain amount of overhead is going to be added to the applications running in the framework. An overarching goal of this research was to enhance the J2EE framework to allow for a more useful and robust handling of exceptions, without compromising the performance of the system. There will of course still be overhead added, but the payoff in terms of useful handing of exceptions, proper separation of concerns, and more robust assemblies should far outweigh any costs.

In order to determine the amount of overhead added to the J2EE framework by the CMEH enhancements, we performed a series of baseline benchmarking tests (see Figure 4.1). The most important factor to maintain was that a minimal amount of overhead should be added in the case where the system developer is not using any CMEH framework features. Ideally, the CMEH framework would not add any overhead at all when its features were not in use.

The benchmarking tests were performed with a variety of mini-component

configurations on a simple, two component system with dummy component methods. In this system, the component methods themselves will add no overhead to the application execution, so accurate tests can be performed to determine the exact overhead added by the CMEH framework. The tests were also performed on a clean JBoss install without any CMEH facilities installed to use as a baseline. Three component methods were tested: the first was an empty `void` method that was used to test method call times, the second returned an `Integer` to test method return times and the third threw an exception on every invocation in order to test method exception propagation times. The reported times represent only the time to invoke the method, not the time required to execute the method body. The time to execute component method bodies is variable and not dependent on any overhead added by the CMEH framework. Each of the tests were run twenty times. The time to propagate exceptions is not reported for the cases where only method-called or method-returned events were being used, since the overhead of the framework would not be any more than when no handlers were deployed in these cases.

Note that the asynchronous handler tests require at least one synchronous handler or else the system will not block at all. Also, the `recover-component-state` handlers require at least one `test-component-state` handler that returns a value of true, or the `recover-component-state method` will never be dispatched, which explains their fairly large invocation times.

Clearly, the CMEH frameworks exhibits a small amount of overhead over a standard JBoss install, even when no event handlers are deployed. This is of course logical, due to the added interceptor in the invocation stack. Particularly, there is more overhead added to method calls when method-called

Server Configurations	Call Time	Return Time	Exception Time
Standard Jboss	2.4 ms	0.8 ms	1.4 ms
CMEH, No handlers	5.2 ms	1.2 ms	1.2 ms

(a) No deployed handlers

Server Configurations	Call Time	Return Time	Exception Time
1 Sync handler	132.4 ms	0.8 ms	N/A
2 Sync handlers	37.0 ms	0.8 ms	N/A
3 Sync handlers	234.2 ms	0.6 ms	N/A
1 Async handler	37.0 ms	1.8 ms	N/A
2 Async handlers	53.8 ms	0.6 ms	N/A
3 Async handlers	76.2 ms	0.4 ms	N/A

(b) method-called handlers

Server Configurations	Call Time	Return Time	Exception Time
1 Sync handler	1.6 ms	36.6 ms	N/A
2 Sync handlers	2.0 ms	63.4 ms	N/A
3 Sync handlers	1.4 ms	141.4 ms	N/A
1 Async handler	2.2 ms	29.2 ms	N/A
2 Async handlers	2.2 ms	35.4 ms	N/A
3 Async handlers	1.8 ms	87.4 ms	N/A

(c) method-returned handlers

Server Configurations	Call Time	Return Time	Exception Time
1 Sync handler	1.8 ms	N/A	42.8 ms
2 Sync handlers	13.2 ms	N/A	60.4 ms
3 Sync handlers	1.8 ms	N/A	51.8 ms
1 Async handler	4.0 ms	N/A	33.8 ms
2 Async handlers	2.2 ms	N/A	53.4 ms
3 Async handlers	2.0 ms	N/A	69.0 ms

(d) method-exception handlers

Server Configurations	Call Time	Return Time	Exception Time
1 Sync handler	1.6 ms	N/A	33.6 ms
2 Sync handlers	1.6 ms	N/A	33.2 ms
3 Sync handlers	1.4 ms	N/A	76.4 ms
1 Async handler	1.6 ms	N/A	25.6 ms
2 Async handlers	10.0 ms	N/A	33.4 ms
3 Async handlers	1.4 ms	N/A	57.0 ms

(e) test-component-state handlers

Server Configurations	Call Time	Return Time	Exception Time
1 Sync handler	10.6 ms	N/A	83.2 ms
2 Sync handlers	1.4 ms	N/A	197.2 ms
3 Sync handlers	1.4 ms	N/A	286.2 ms
1 Async handler	2 ms	N/A	75.6 ms
2 Async handlers	1.6 ms	N/A	121.4 ms
3 Async handlers	1.4 ms	N/A	163.2 ms

(f) recover-component-state handlers

Table 4.1: Latency times for various CMEH event handler configurations

event handlers are deployed. This is logical due to the fact that JBoss interceptor stack contains more interceptors processing method calls, compared to method returns and exception propagation. The added overhead for the recover-component-state event handlers is an acceptable cost due to the fact that these events occur irregularly, only when the component is left in an invalid state.

Although these overheads are significant, these factors are quite minimal when compared to the actual execution time of a method. The time to simply call and return from the method is insignificant when compared to the execution time of a substantial method body.

The latency for the asynchronous handlers is minimal, but the synchronous handlers have a fairly significant overhead. In this test it is partly due to the fact that the asynchronous handlers are not performing any processing, so they don't interfere with one another and therefore the synchronous handler blocking on them does not have to block very long before the asynchronous components have completed their processing. This aside, the performance overhead of the synchronous listeners is significant and should probably be reduced. Currently all of the handlers are implemented using JMS which adds a fair amount of overhead to the synchronous handlers to ensure that they block and that they execute in sequence. JMS could most likely be replaced in the synchronous event handlers by a simpler Java `Event`-based system, which would reduce the overhead, although it would reduce the symmetry of the implementation.

Another important overhead to examine in the application server with CMEH facilities is memory usage (Figure 4.2). The memory usage of the CMEH-enabled application server was compared to that of a standard JBoss install.

Configuration	RAM Usage (K)
Standard JBoss Memory Usage	270,044
CMEH-enabled Jboss Memory Usage	274,564

Table 4.2: Memory usage in various JBoss configurations

While there is a four megabyte memory usage increase, this is less than a 2% increase.

Chapter 5

An Example Application

5.1 Application Architecture

A COTS-based application was developed by the Software Performance Architecture group at the Nokia Research Center in Burlington, MA. This application was a prototype for demonstrating a new HTML processing method being developed at NRC. The prototype was made up of a variety of components, both COTS and proprietary, as well as components from elsewhere in the Nokia organization for which source code was either not available or not well understood by the developers of the prototype application. The architecture of the prototype was modified slightly for this research in order to fit it into the EJB model (see Figure 5.1), however the code of the components themselves was unchanged, and all components were treated as COTS components.

The prototype has a web browser as a client front-end. Via the browser, users connect to a `HTTPServlet` running on a J2EE application server. The users input a URL for a website, which sets the application in motion. The

application fetches the website from the given URL, performs the necessary processing and returns the modified HTML back to the users browser, which in turn displays the page.

The servlet forwards the request in the form of a URL string to the `HTMLTransformer` component, which is the main component of this system. The exact details of this component are currently under patent review and can therefore not be disclosed. The component then forwards the URL string to the HTML parsing and rendering component, a commercial component called ICEBrowser^{TM} by ICESoft Technologies. This Java-based component can be used as either a front-end or server-side solution. It is capable of parsing and rendering HTML and can be used as a base for a substantial web browser. In the case of this system, the component was not used as a widget, so the rendering was only a layout process, where the results were not actually rendered to any display. The ICEBrowser component connects to the Internet via an `HTMLConnector` component. This component receives the string URL and performs an `HTTP-GET` to retrieve the HTML data at the given URL. In then returns the stream of HTML to the ICEBrowser component. This component then parses and renders the HTML and creates an augmented Document Object Model (DOM) tree. This tree is a hierarchical representation of an XML (or HTML) file, where the first tag of the file is the root of the tree, and the branches and leaves are filled out accordingly. The DOM tree is used by the ICESoft browser for rendering purposes. It traverses the tree and renders each element based on the HTML specification. The ICESoft component augments this DOM tree with additional layout information for its rendering purposes, and that information is crucial to the `HTMLProcessor` component.

Once the `HTMLProcessor` component gets the augmented DOM tree, it

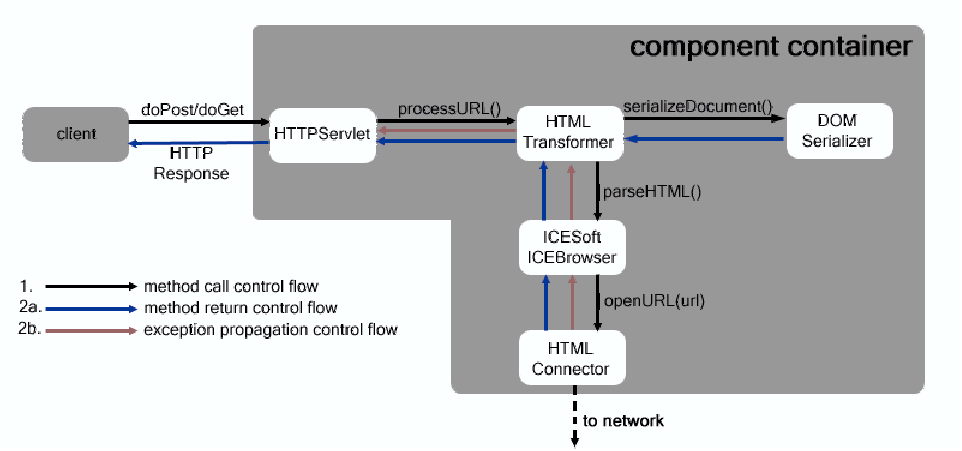

Figure 5.1: An example use case leveraging the CMEH Framework

performs its necessary processing to transform the HTML. The component then needs to convert the tree back into HTML to send to the client's browser. This is accomplished by passing the DOM tree to the `DOMSerializer` component, a proprietary component created elsewhere in the Nokia organization. This component traverses the DOM tree and converts it back into browser-readable HTML. The HTML is sent to the user's browser via the servlet.

However, when these components are assembled as is, there are several major problems with the system and it doesn't function in a robust fashion. In fact, it is very difficult to find a URL that the system can successfully process. One such problem involves sending the URL string to the `HTTPConnector` component. If this string is not properly formatted with the `http://` prefix, the URL lookup will fail. The next problem also involves the `HTTPConnector` and the ICEBrowser component. In certain situations, the ICEBrowser will be parsing an HTML stream from the `HTTPConnector` and it will receive certain JavaScript commands (such as `onLoad()`), that will cause it to make another request to the `HTTPConnector`. If these re-

quests come too rapidly, the `HTTPConnector` will throw an exception and the request will fail.

Another problem occurs once the parsing component has created a DOM tree. Since the HTML data being sent back to the browser is not coming directly from the web server at the original URL, the addresses for resources, such as images, cannot be resolved by the browser. As a result, the resource URLs in the DOM tree must be converted into absolute addresses. A final problem occurs when the DOM is sent to the `DOMSerializer` component. When the ICESoft parser creates the DOM tree, it automatically converts HTML special characters into the ASCII values. This creates a problem when the DOM is converted back into HTML because the ASCII characters do not display properly in most conventional browsers. Without addressing all of the aforementioned concerns, this system does not function at an acceptable level of correctness or reliability.

5.2 Correcting the Exceptional Behavior

In order to perform an accurate evaluation of the CMEH framework, these errors were first corrected using "glue code" wrapper components and then using CMEH mini-components. The two solutions were compared on a number of criteria: number of lines of Java written, amount of XML written, performance, and correctness. The CMEH version was also compared for performance and correctness to the base application without any correction code.

To correct the shortcomings of this application using the CMEH framework, four CMEH mini-components were created. The first was a `method-called` event handler that is responsible for ensuring the proper for-

matting of URLs. The URLs require a protocol prefix for both the ICESoft parser and the `HTMLConnector` components. The code and XML deployment configuration for the `method-called` event handler can be seen in Figure 5.2 and Figure 5.3.

```
1  public class MethodCalledURLChecker
2                          implements MethodCalledEventHandler {
3
4          public Object[] handleMethodCalledEvent(
5                  Invocation arg0,
6                  ExceptionEvent arg1,
7                  MethodCalledEventContext arg2)
8                  throws Exception {
9
10                 Object[] args = arg2.getArguments();
11                 String url = (String)args[0];
12
13                 if(url.indexOf(':') == -1) {
14                         url = "http://" + url;
15                 }
16                 return new Object[] {url, args[1], args[2]};
17         }
18 }
```

Figure 5.2: CMEH method-called mini-component for verifying URL requests

The second mini-component is a `method-returned` event handler that translates all of the URLs returned as part of the DOM from the HTML parser component into absolute addresses (Figure 5.4).

Another mini-component was created to monitor the `method-called` event of the `serializeDOM()` method of the `DOMSerializer` component (see Figure 5.5 and Figure 5.6). This mini-component's task is to translate any special ASCII characters into their corresponding HTML characters.

The final CMEH mini-component was created in order to handle method

```
<exception-handler>
        <method>
                <interface-name>
                        com.nokia.minicooper.HTMLParserRemote
                </interface-name>
                <method-name>parseHTML</method-name>
        </method>
        <method-called
        class="com.nokia.minicooper.MethodCalledURLChecker"/>
        <method-returned
class="com.nokia.minicooper.MethodReturnedAbsoluteURLs"/>
</exception-handler>
```

Figure 5.3: CMEH XML configuration for method-called mini-component for checking URLs and method-return mini-component for resolving relative URLs

```
public class MethodReturnedAbsoluteURLs
                implements MethodReturnedEventHandler {

        public Object handleMethodReturnedEvent(
                Invocation arg0,
                ExceptionEvent arg1,
                MethodReturnedEventContext arg2)
                throws Exception {

                DDocument doc =
                        (DDocument)arg2.getReturnValue();

                // resolveRelativeURLs is a private method to
                // iterate the DOM and resolve the URLs
                this.resolveRelativeURLs(doc);
                return doc;
        }
}
```

Figure 5.4: CMEH method-return mini-component for resolving relative URLs

```
1  public class MethodCalledHTMLTranslator
2                          implements MethodCalledEventHandler {
3
4          public Object[] handleMethodCalledEvent(
5                  Invocation arg0,
6                  ExceptionEvent arg1,
7                  MethodCalledEventContext arg2)
8                  throws Exception {
9
10                 Document doc =
11                         ((Document)arg2.getArguments()[0]);
12
13                 // translate the necessary html
14                 NodeList all = doc.getElementsByTagName("*");
15
16                 for(int i = 0; i < all.getLength(); i++) {
17
18                         Element e =  (Element)all.item(i);
19                         Node n = e.getFirstChild();
20
21                         while(n != null) {
22
23                                 if(n.getNodeType() ==
24                                 Node.TEXT_NODE &&
25                                 !isInHeadOrScript(n)) {
26
27         // encode is a method in
28         // org.htmlbrowser.translate.Translate
29         n.setNodeValue(Translate.encode(((Text)n).getData());
30
31                                 }
32                                 n = n.getNextSibling();
33                         }
34                 }
35                 return new Object[]{ doc };
36         }
37 }
```

Figure 5.5: CMEH method-called mini-component for translating special ASCII characters to HTML

```
<exception-handler>
        <method>
                <interface-name>
                com.nokia.minicooper.DOMSerializerRemote
                </interface-name>
                <method-name>serializeDocument</method-name>
        </method>
        <method-called
class="com.nokia.minicooper.MethodCalledHTMLTranslator"/>
</exception-handler>
```

Figure 5.6: CMEH XML configuration for method-called mini-component for translating ASCII characters to HTML

exceptions from the `openURL()` method of the `HTMLConnector` component (Figure 5.7 and Figure 5.8). When this component method throws exceptions, it is generally due to either a network connection problem, or the aforementioned JavaScript thrashing problem. Rather than simply propagate an exception back to the client, an error HTML page is generated by the CMEH `method-exception` handler, and is sent back to the client. With this approach, the error HTML page may be passed to the other components in much the same way as the requested HTML content would have been.

In order to create the glue code version of the application, three new EJBs were created in order to wrap the existing html parser, connector and serializer EJB. The reason these glue code wrappers needed to be full EJBs was due to the fact that the existing components were being treated as COTS components. In order to allow the glue code component to be transparently inserted into the application, the glue code components needed to implement the same interfaces as the original components. Then, the JNDI names of the original components were modified and the glue code components claimed

```
1  public class MethodExceptionErrorPageGenerator
2                  implements MethodExceptionEventHandler {
3
4          public Object handleMethodExceptionEvent(
5                  Invocation arg0,
6                  ExceptionEvent arg1,
7                  MethodExceptionEventContext arg2)
8                  throws Exception {
9
10                 // return a "default error page"
11                 String errorPage = "<HTML>";
12                 // code clipped
13                 errorPage += "</HTML>";
14                 return new ByteArrayInputStream(
15                         errorPage.getBytes());
16         }
17 }
```

Figure 5.7: CMEH method-exception mini-component for generating default return value error pages

```
1  <exception-handler>
2          <method>
3                  <interface-name>
4                  com.nokia.minicooper.HTMLConnectorRemote
5                  </interface-name>
6                  <method-name>openURL</method-name>
7          </method>
8          <method-exception
9  </exception-handler>
```

Figure 5.8: CMEH XML configuration for method-exception mini-component responsible for generating default return values

the original JNDI names of the components. This way, when a component attempted to look up the original components, they would instead receive a reference to the glue code correction component. Then the correction components could perform any necessary modifications and then relay the method call to the original components.

5.3 Experimental Results

The number of additional lines of code (LOC) and lines of XML required to implement the two versions were counted (Figure 5.1).

Line	Glue Code Version	CMEH Version
Lines of Code (LOC)	165	115
Lines of XML	81	22

Table 5.1: LOC count in the CMEH mini-cooper version vs. glue code in an example use case

Clearly, the glue code version requires substantially more Java code and XML to implement, even with a relatively simple system like this. It can be expected that this fact would be exacerbated in a larger system.

Another interesting statistic for this test case was the amount of time required to develop the glue code EJBs and the CMEH mini-components. To correct the systems behavior with the glue code took a programming effort of 59 minutes. The CMEH version took only 45 minutes. Although this is quite a small and specific use case, the differences in development time is still notable. Furthermore, due to the modularity of the CMEH mini-components, there is an opportunity for reuse that is not present in the glue version.

Several tests were performed on the various configurations of the system based on a list compiled at Nokia of the most frequently visited website on the Internet. Each of the three configurations (the base application, the correction EJB version and the CMEH version) were repeatedly tested for performance on the top four websites from the list. Since this application utilizes data accessed across the Internet, there is a large amount of possible variations in performance costs due to network latency. The performance tests were run in order to determine if the CMEH version of the system could run in approximately the same latency as the glue code version, taking into a account the fact that a great deal of the latency is actually due to network latency. The performance statistics are present in Table 5.2.

Website	Standard JBoss	Glue Code	CMEH
www.yahoo.com	340.0 ms	812.4 ms	1055.2 ms
www.msn.com	873.8 ms	1128.7 ms	1842.9 ms
www.alexa.com	877.9 ms	890.4 ms	897.7 ms
www.passport.net	934.4 ms	954.4 ms	1038.9 ms

Table 5.2: Processing times for all approaches in an example use case

In all but one case the CMEH version suffered from the worst performance. However, the values were on average only 18% slower than the glue code version and 61% slower than the base application. While this should be improved, it is an acceptable cost for the added benefits of the framework.

Another set of data was gathered based on how correctly each version of the application loaded each of the top sites. The three different systems were used to load each of the sites, and the correctness was based on whether or not the loaded pages contained rendering, hyperlink, image, and scripting errors, illustrated in Figure 5.3.

Error Description	Base Application	CMEH Version
Pages Loaded Correctly	9	40
Rendering Errors	41	7
Scripting Errors	7	9
Image Errors	14	0
Hyperlink Errors	14	0
Complete Failures	34	32
Total Errors	110	48

Table 5.3: Generated errors in the CMEH version vs. base application in an example use case

The CMEH version and the glue code methodologies produced the same results because their methods for correcting the erroneous behavior of the system were identical, although the architectures were of course drastically different. While all of the systems suffered from reasonably high rates of failure, this is not the important statistic to notice. Some of failures were due to the sites being down and failures of the COTS HTML parser component, which could not be handled in the system. Furthermore the serialization of the DOM back into HTML is not a trivial task and the `DOMSerializer` component is not always able to create HTML that can be rendered on all browsers. However, what should be taken from these results is how dramatically a few simple CMEH components can improve the behavior of the system. The two additional scripting errors that occurred in the CMEH version of the application were due to the fact that those pages failed entirely in the base application.

This use case provided a relatively conclusive set of results for the CMEH framework. The performance overhead for the CMEH version of the appli-

cation was very minor when compared to the glue code version. The CMEH version created a more robust and correct system with a relatively small amount of code and XML configuration, when compared to the glue code version. Furthermore, the CMEH version provides a much greater separation of concerns. The CMEH mini-components are much more loosely coupled with the EJB components than the glue code EJBs are. If one of the original EJBs was upgraded or modified, it could potentially require a major reworking of the glue code components, whereas the CMEH mini-components could be updated in a simple fashion. Furthermore, the CMEH version allows the mini-component to be dynamically swapped out, whereas removing the glue code components would require a full redeployment of the system, which would of course involve first taking the system down entirely.

Chapter 6

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). AOP is a means of extending a programming language with the ability to separate concerns by cutting across the standards units of modularity. A great deal of research has gone into using AOP as an effective means of handling exceptions in software systems [14]. In fact, the initial architecture for the container-managed exception handling model involved using AspectJ^{TM} rather than the container interceptor method. However, using AspectJ proved to be insufficient due to several limitations of the language. First, the AspectJ language requires that any class that is to be crosscut be compiled in conjunction with the aspect code by the AspectJ compiler. This is an unreasonable requirement when working with COTS components, as the source code is often not available for recompilation. A second drawback is that AspectJ advice cannot throw a subclass of the class `Exception` that is not thrown by the method it is monitoring, making exception translation difficult and cumbersome. However, much like this interceptor approach, AOP for excep-

tion handling stresses the detangling of exceptional code, as well as support for multiple configurations, incremental development and dynamic reconfiguration [14]. The way in which interceptors are leveraged in the container-managed exception handling system is quite similar to way in which `around` advice is used in AspectJ. It gives the developer the ability to crosscut in order to execute before and after the method invocation, as well as the ability to decide whether the method should proceed at all.

The focus on the separation of concerns in the container-managed exception handling model is also influenced by research being done in the field of multi-dimensional separation of concerns [26]. The hyperslicing mechanism that is a part of the Hyper/J^{TM} tool ([25]) is a large scale abstraction of the type of crosscutting mechanism used in this component container research. Hyper/J uses these hyperslices to separate multiple simultaneous concerns in several dimensions, allowing for incremental development and composition.

The asynchronous portion of this research and its relation to proper separation of concerns is related to the Actor model of parallel and distributed components [1]. The Actor model provides an asynchronous, non-blocking, parallel model for distributed component-based systems. In this model, there is no notion of a caller of a component method being responsible for handling the exceptions thrown by the method that its calling. This is because in the Actor model, calling method involves sending a message to a remote component and that message is stored in a queue. The component services the request when it has an opportunity, and the caller of that method does not block. Therefore, by the time the called component finally services the request, the caller has moved into a state where its no longer capable of handling an exception. As a result, the exceptional behavior must be handled by a third component that knows how to correctly handle the exceptional

behavior. This is of course directly related to the CMEH framework.

Research in augmenting JBoss containers was done at the MITRE Corporation by Gary Vecellio et al. [28]. 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. With these assertions, the developers of a system will be able to enforce their assumptions about the quality attributes of the system. If it is essential that a particular component return its output within a certain time constraint, an assertion is added to the container of that component. If the component does not return a value in the given time, the assertion fires, alerting the operator that the quality attribute is not being met. The system developer can then modify or replace the component in order to meet the necessary quality attributes of the system. These features are configurable by the system developer at deploy time via the EJB XML deployment descriptor. This research has been highly influential in this container-managed exception handling model. More recently, additional work was done in the domain of policy enforcement in component containers [27]. A policy is a constraint on a system-wide property. It is fairly easy to write policy enforcement in a non-component-based application, however the problem becomes significantly more difficult in a component-based systems, since the services and components are developed separately from one another. Furthermore, it is important to be able to dynamically change the policies for the system without disrupting the system itself. This is particularly important for systems that require robust certification. The proposed scheme for policy enforcement involves separating the policy from the business logic by providing policy enforcement in the JBoss interceptor stack. In this way, interceptors can enforce policies based on the states of the components and can be dynamically shuffled in and out until

an appropriate policy is found.

Research in dynamically generating containers is being conducted at the Ohio State University by Jason Hallstrom and Nigamanth Sridhar [21]. They state that traditional encapsulation techniques fail with component-based systems since a system developer cannot simply place each concern into its own component. This is because of the fact the boundaries dictated by one concern may overlap with the boundaries of other concerns. In order to deal with these issues, there are a number of aforementioned solutions like AOP and N-Degree separation of concerns. However, this research focuses on the component container. Rather than using the distributed EJB container, these researchers created a new container model based on the Service Facility or Serf pattern. In this pattern, components interact with all services and data objects through the Serf. The component requests a service from the Serf and the Serf provides the components with the necessary data objects it needs. The researchers view the Serf container as a parameterized component; one where some, but not all of the services are fixed and can be dynamically plugged and parameterized. Thinking of the container as a parameterized component allows for standard reasoning and proof techniques to be applied to containers, something that is a major detriment to current containers. A Service Facility Adapter Tool (SFAT) was created to dynamically generate Serf containers to encapsulate services provided by Microsoft C# libraries.

A recent research project summarized the current exception handling practices and issues regarding them in the EJB framework [20]. The researchers classified all of the components in J2EE into three classes: CS components (contract-based interactions with synchronous communications), CA components (contract-based interactions with asynchronous communica-

tions) and E components (event-based interactions with synchronous communications). Typical EJB session and entity beans fall into the first category, while message-driven beans are CA components. Regular JavaBeans are an example of E components. The main concern of this research was that the exception handling abilities for CA components are extremely limited, as the synchronous components can rely on typical Java try/catch constructs. With MDBs, the sender of a JMS message can be notified of an error, but no exception is propagated back. The authors of this research specify four conditions that they feel are necessary for properly handling exceptions in asynchronous components. They are 1) the place where exceptions are most usefully handled are at the caller, so a mechanism for getting the exception back to the caller is necessary, 2) while asynchronous components are not supposed to synchronized, mechanisms need to be in place to synchronize the component during exception handling situations, 3) a means to collect concurrent exceptions is needed, and 4) an exception mechanism is needed for broadcast requests. This research is still in a fledgling state and does not suggest augmentations to the J2EE framework to support these necessities.

Recently several other research projects have begun focusing on exception handling in component-based systems. One such project is being conducted at the Instituto de Computao at the Universidade Estadual de Campinas, Brazil by Ricardo de Mendona et al [8]. This research relies on a concept known as compositional contracts, based on the Coordinated Atomic Action (CA Action) scheme. A CA action is an atomic action made up of several component participants. This means that methods from the interfaces of several components are combined into a single atomic action. Using compositional contracts based on these CA actions, exceptions in component systems can be detected and potentially recovered from. All the participants

in the coordinated action are communicated with via an asynchronous framework. If a single participant raises and exception, the framework receives the exception and attempts to recover in one of two ways. The first way, known as rolling forward involves attempting to create a degraded return response rather than propagating the exception itself. This is similar to intercepting an exception in the CMEH framework and returning a default value rather than propagating the exception. If rolling forward fails, the framework attempts to roll back the atomic action, as specified by the contract. If this succeeds, an abort exception is propagated back to the caller. If the rolling back fails, a failure exception is propagated. If more than one participant in a CA action throw exceptions, the framework attempts to combine the two exceptions into a single exception and then recovery can proceed as usual. This work is extremely interesting and relevant to the CMEH framework research, however it is not immediately clear what sort of assumptions the authors are making about the involved components. It is a complicated problem to roll back the components in to legal states, and it's not immediately obvious that current COTS components would be capable of being recovered simply be contract specification. Furthermore, not all COTS components can be treated as thread-safe asynchronous components. A great deal of care must be taken to make components suited for parallelism. However, in terms of designing proprietary components for use in this particular framework, the benefits seem quite clear and excellent.

Similar CBSE exception handling research is also being conducted by Alexander Romanovsky at the University of Newcastle upon Tyne, United Kingdom [17]. Dr. Romanovsky's framework involves the application of a three-tiered architecture. First, components are wrapped in a wrapper that is used to handle local errors and exceptions. This wrapper catches all returned

error codes and exceptions, as well as testing a set of predicates in order to determine if a component operation has resulted in an error. Recovery is then attempted at the local level. The approach for this is very similar to component state recovery in the CMEH framework; however the author implies that this can easily automated with COTS components. This is not made entirely clear by the proposed solution. The second tier of the framework is similar to the CA action approach. All of the actions of the system are grouped into atomic dynamic actions. This is done in order to facilitate and contain erroneous situations. The final tier involves applying an exception handling layer to the atomic actions. If the error cannot be handled at a local level, it is attempted to be handled at an action level. Here the author concedes to the fact that state recovery can be increasingly difficult with todays complex software systems, and there is a need for recovery of the application at a system level. For the most part, this research seems to be assuming an ideal set of COTS components which can be grouped easily into useful atomic actions and that provide their own means for recovery. This framework has not been fully implemented, so perhaps some of these assumptions will change once the proposed CORBA implementation has been created. More recently, a use case based on a simplified version of this framework (only the component wrappers) has been created with Simulink (part of the MATLAB product) [2]. This use case is a simulation of a COTS Proportional, Integral and Derivative (PID) controller for a steam boiler system. The researchers created a protective wrapper around the PID components in an effort to test forward recovery techniques. Some assumptions were made to simplify the scenario, such as the instantaneous availability of variable values. The protective wrapper tested the values of several component properties and also caught all exceptions thrown by the component. The error recovery strategy

was encapsulated in three error handlers. The first handler's technique was essentially to notice an erroneous value in the component and do nothing about it in the hopes that after a brief amount of time, the value would be corrected. If this was unsuccessful, then the second handler would attempt to take the value from the controller and adjust it to a legal value, while notifying the operators of the discrepancy. If an exception arose that could not be handled by either of these two methods, the third handler shutdown the system and notified the operators. While a fairly simple example, this research does provide some useful results and insights. First, the component being used is a real COTS component, giving the results more viability than if a proprietary component were used. Secondly, an important result of this work pertained to the complexity of the wrapper itself. That result was that, in most cases, simpler is better and that the wrappers and framework that the component is running in should not contribute to an increase in failure in the entire system.

Chapter 7

Conclusions and Future Work

Component-based software engineering (CBSE) has shown great promise in recent years for greatly reducing both the cost and the time-to-market for software systems. Building systems from commercial components allows system developers to shop for prefabricated binary implementations of desired functionality, rather than paying a developer to create similar software, greatly reducing development time and saving considerable money. Component-based systems also promote more robust software systems, since components will, in theory, be more highly distributed and tested.

However, to this point, component-based development has not lived up to the expectations of the software industry. One of the main reasons for this is that the behavior of systems assembled from components cannot be accurately predicted. When developing software systems, certain extra-functional properties of the system need to be ensured. However, this has proven to be a very difficult task when dealing with COTS components, due to their black-box nature.

Component containers have emerged as a step in the right direction to-

wards solving the problem of predictable assembly. By applying a set of services in the container (i.e. security policy enforcement, transaction management), certain quality attributes can be better ensured. Implementing these services in the container allows component developers to concentrate on business logic, providing a good separation of concerns. However, there are still countless concerns regarding predictable assembly that have not been properly addressed. One such concern pertains to the handling of exceptions in component-based systems. Because component developers cannot be aware of the other components their software will interact with, they cannot possibly predict the exceptional behavior of those components. Furthermore, with current standards, exceptions cannot be handled in an application-specific way. Because of the deficiencies with current exception handling techniques, code for handling exceptions in component-based systems is either generic (and therefore mainly useless), or a mess of code tangle generated by a series of `try-catch` constructions. Before component-based systems are made more robust, predicting the quality attributes of components in the system cannot be done accurately.

This research suggests augmenting the component container in order to allow exceptions to be handled in a modular, application-specific way. The Container-Managed Exception Handling (CMEH) framework allows system developers to create exception handling mini-components that handle exception behavior in the system. These mini-components handle exceptional behavior in three phases. First it allows component method call arguments and return values in order to prevent exception in the system. This serves as a solution to the partial semantic matching problem. The second phase allows the system developer to intercept exceptions thrown by component before they propagate to the calling component. By intercepting the exceptions,

system developers can translate the exceptions to useful types of exceptions that the calling component can handle or stop the propagation of exceptions and instead return useful default values. The third phase of the framework provides the event handling mini-components with an attempt to test the state of components after exceptions occur in the system. If the states of components are invalid, the CMEH framework has an opportunity to recover the state of the components with developer-deployed mini-components. This crosscutting framework provides an excellent separation of concerns since the exception handling code is well removed from the business logic of the components. The CMEH framework leads to robust system functionality, reducing the number of errors in the execution of the system, and thus the mean-time-to-failure(MTTF). Reducing the variations in the MTTF and reliability of the system allows for a more meaningful and accurate prediction of other quality attributes. Attempting to accurately predict the performance or availability of a non-robust system is much less possible and valuable than predicting the extra-functional properties of a system that has a more robust functionality. By properly handling the exceptional behavior of the system, it is possible to more accurately predict other extra-functional properties of the system.

Benchmarking tests on the performance of the framework show that the amount of overhead added to the framework is nearly minimal, suggesting that the benefits of framework outweigh any performance overhead. These claims are further reinforced by a real use case developed at the Nokia Research Center in Burlington, MA. This prototype application involved processing HTML via several COTS components. Using the CMEH framework, several deficiencies in the application were corrected, creating a more robust system. The problems in the application were also corrected using a more

conventional "glue code" wrapper approach. The tests concluded that the CMEH version produced a simpler, more modular, and more robust system. Furthermore, the CMEH version was drastically more maintainable and updateable, whereas the glue code version was highly coupled to the components.

Despite the positive results of the CMEH results, there are still a number of improvements that should be made to the framework. First of all, the overhead for adding a synchronous mini-component is too large at this point. Since it is based on the same JMS `Event` model as the asynchronous handlers, the overhead is similar. However, it would more than likely suffice to base the synchronous handlers on a simple architecture based on the Java event model. Since synchronous handlers will no doubt be the more frequently used, it is important to minimize their overhead.

Furthermore, while the current asynchronous model provides a nice multi-threaded approach, it would be beneficial to distribute and parallelize the exception event handlers. Currently, it would be possible to deploy the mini-components on separate machines and then manually connect the handlers on one machine to an application on another via JNDI, this would be a bit cumbersome. By augmenting the deployment options in the XML deployment descriptor, remote deployment could be specified and automated (Figure 7.1). For small exception handling tasks, this would no doubt be overkill since the network latency would far outweigh the benefit of the parallel execution, but for large tasks, such as recovering component states, this may prove to be a very powerful option.

There are also several tasks in the CMEH framework that could be further automated in order to minimize the amount of code a system developer has to write for the mini-components. Simple test-component-state facilities that

```
<exception-handler>
        <method>
                <interface-name>MyEJBRemote</interface-name>
                <method-name>method1</method-name>
                </method>
        <test-component-state class="com.tufts.cmeh.ts1"/>
        <recover-component-state
                class="com.tufts.cmeh.rs"
                remote="true">
                <remote-server ip-addr="192.168.1.2"
                        proto="ssh">
                        <user-name>ksimons</user-name>
                        <password>abc123</password>
                        <dir>
                                ~/jboss/server/default/deploy
                        </dir>
                </remote-server>
        </recover-component-state>
</exception-handler>
```

Figure 7.1: CMEH remotely deployed recover-component-state handler

check public properties via "get" methods and compare the results to values provided in the XML deployment descriptor could be added, making the test for invalid states far simpler.

Perhaps the most compelling research direction this project produced is that of recovering component states after an exception. A great deal of work is currently being done to make component methods atomic in order to automatically recover systems after faults [8, 17]. However, while it is fairly easily to recover a single component after an exception, it is increasingly difficult to handle situations where a "ripple effect" causes other component to also be in invalid states. This becomes a reconfiguration problem, where one or more components must be swapped out or reloaded in order to bring the system back into working order. Current transaction techniques are insufficient for solving this problem. First of all, the current J2EE transaction model is not hierarchical. This means that if a component calls another component and the second component's method completes successfully, the component's state will be committed to the database regardless of whether or not the method of the calling component completes successfully. Furthermore, hierarchical transaction systems introduce an unacceptable amount of overhead to the system.

Analyzing a running system to determine what interdependencies exist between components is a very difficult research problem. Further research will be done at Tufts University by the author in order to develop an Architecture Dependency Language that allows for the specification of component dependencies, as well policies for exception recovery. Static and dynamic control and data dependence analysis will be run over the component-based systems in order to properly determine component dependencies for use in error recovery.

Bibliography

[1] G. Agha and W. Kim. Actors: a Unifying Model for Parallel and Distributed computing. *Journal of Systems Architecture*, 1999.

[2] T. Anderson, M. Feng, S. Riddle, and A. Romanovksy. Error Recovery for a Boiler System with OTS PID Controller. *Proceedings of the Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and new Programming Paradigms workshop, Darmstadt, Germany*, 2003.

[3] M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillman, and A. Watson. Serious Specification for Composing Components. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[4] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume I: Market Assessment of Component-based Software Engineering. *Technical Report CMU/SEI-2001-TN-007, Software Engineering Institute*, 2000.

[5] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical Concepts of Component-Based

Software Engineering, 2nd Edition. *Technical Report CMU/SEI-2000-TR-008. Software Engineering Institute*, 2000.

[6] A. Bertolino and R. Mirandola. Towards Component-Based Software Performance Engineering. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[7] B. Councill. Third-Party Certification and Its Required Elements. *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction (CBSE4), Toronto, Canada*, 2001.

[8] R. De Medona da Silva, P. Asterio de C. Guerra, and C. Rubira. Component Integration using Compositional Contracts with Exception Handling. *Proceedings of the Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and new Programming Paradigms workshop, Darmstadt, Germany*, 2003.

[9] G. Heineman. Integrating Interface Assertion Checkers into Component Models. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[10] P. Inverardi and M. Tivoli. Correct and automatic assembly of COTS components: an architectural approach. *Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly (CBSE5), Orlando, Florida*, 2002.

[11] P. Kallio and Eila Niemel. Documented Quality of COTS and OCM Components. *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction (CBSE4), Toronto, Canada*, 2001.

[12] M. Larsson, A. Wall, C. Norstrom, and I. Crnkovic. Using Prediction-Enabled Technologies for Embedded Product Line Architectures. *Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly (CBSE5), Orlando, Florida*, 2002.

[13] K. Lau. Component Certification and System Prediction: Is there a Role for Formality? *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction (CBSE4), Toronto, Canada*, 2001.

[14] C. Lopes, J. Hugunin, M. Kersten, M. Lippert, E. Hilsdale, and G. Kiczales. Using AspectJ for Programming the Detection and Handling of Exceptions. *Proceedings of the ECOOP Exception Handling in Object Oriented Systems Workshop*, 2000.

[15] J. McGregor, J. Stafford, and I. Cho. Measuring Component Reliability. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[16] P. Mehlitz and J. Penix. Design for Verification: Using Design Patterns to Build Reliable Systems. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[17] A. Romanovsky. Exception Handling in Component-Based System Development. *Proceedings of the 25th International Computer Software and Application Conference (COMPSAC 2001), Illinois*, 2001.

[18] S. Shenoy. Best practices in EJB exception handling. *http://www-106.ibm.com/developerworks/java/library/j-ejbexcept.html*, 2002.

[19] M. Sitaraman. Compositional Performance Reasoning. *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction (CBSE4), Toronto, Canada*, 2001.

[20] F. Souchon, C. Urtado, S. Vauttier, and C. Dony. Exception handling in component-based systems: a first study. *Proceedings of the Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and new Programming Paradigms workshop, Darmstadt, Germany*, 2003.

[21] N. Sridhar and J. Hallstrom. Generating Configurable Containers for Component-Based Software. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[22] J. Stafford and J. McGregor. Issues in Predicting the Reliability of Composed Components. *Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly (CBSE5), Orlando, Florida*, 2002.

[23] J. Stafford and K. Wallnau. Predicting Feature Interactions in Component-Based systems. *Proceedings of the ECOOP Workshop on Feature Interaction of Composed Systems*, 2001.

[24] C. Szyperski. *Component Software: Beyond Object-Oriented Programming: Second Edition*. ACM Press: New York, 2002.

[25] P. Tarr and H. Ossher. Hyper/J: Multi-Dimensional Separation of Concerns for Java. *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[26] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[27] G. Vecellio and W. Thomas. Infrastructure Support for Predictable Policy Enforcement. *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering*, 2003.

[28] G. Vecellio, W. Thomas, and R. Sanders. Containers for Predictable Behavior of Component-based Software. *Proceedings of the Fifth ICSE Workshop on Component-Based Software Engineering*, 2002.

[29] A. Zaremski and J. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, pages 333–369, 1997.