Aspect-Oriented Programming with Enterprise JavaBeans

Jung Pil Choi
Department of Computer Science and Engineering
Korea University
Seoul, Korea
jpchoi@psl.korea.ac.kr

Abstract

Enterprise JavaBeans (EJB) provides special functionalities such as transaction, persistence, location transparency, and security. These functionalities can be considered as aspects and EJB can be regarded as an Aspect-Oriented Programming (AOP) environment. However, its architecture has no extensibility and flexibility to add or modify aspects, so it is not considered as a general AOP environment and hard to be applied to diverse domains. In this research, a new EJB server, named AES, is designed and implemented. An aspect exists independently of a container, and can be added and updated as needed. A container is changed into a generalized metaobject supporting more aspects in a flexible and extensible way. Also, an EJBObject is changed as a reference aspect supporting various network protocols. To prove flexibility and extensibility of AES, a testing application and user-defined aspects are developed. Finally, overhead for supporting AOP in EJB is measured.

1. Introduction

Enterprise JavaBeans (EJB) [1] is a popular network middleware and application server that provides non-functional aspects [9] such as transaction, persistence, location transparency, and security. It successfully relieves domain specific programmers from additional burdens for those aspects.

However, we found out that EJB is not adequate for diverse domains such as embedded systems, desktop development environment, etc. The architecture and most of its implementations are too monolithic to add, remove, and configure a module in a system. In the viewpoint of Aspect-Oriented Programming (AOP) [9], the architecture of EJB has no extensibility to add other aspects such as logging, debugging, and load-balance. Also, EJB does not permit the existing aspects and its policies to be modified. To solve these problems, an EJB server should be developed with the architecture that can support more general AOP concepts, and with that architecture the EJB server itself will be more flexible and extensible.

The remaining sections are organized as follows: Section 2 discusses related work. Section 3 presents elements of AES and related design-issues. Section 4 discusses an implementation of AES in the details. Section 5 describes expected performance overhead compared with the JOnAS [4] EJB server. Section 6 presents our conclusions.

2. Related Work

OpenWebServer [15] is developed to support the dynamic adoption of functionality such as introduction of additional protocols, modification of execution policies, and tuning system performance. This is achieved by specifying and coordinating metaobjects that represent various aspects within the web server. For example, `ThreadPerSessionAcceptor` metaobject can be switched to `ThreadPerRequestAcceptor` metaobject at run-time by reification from the base-level.

OpenCorba [10] is a CORBA implementation based on a reflective extension of the Smalltalk environment. OpenCorba provides users with a mechanism to adapt dynamically the representation and the execution policies of the ORB. For example, it can change its remote invocation mechanism. The OpenCorba Interface Definition Language compiler automatically generates the proxy class on the base-level of the application and associates the proxy class with the meta-class `ProxyRemote` in charge of calling the real object. The remote invocation remains transparent for the client while the `ProxyRemote` is changed into another class during run-time.

AspectJ [11] is a general-purpose aspect-oriented extension to Java. AspectJ extends Java with a new module, called an *aspect*. Aspects have the ability to cross-cut classes, interfaces, and other aspects. This means that the code in a single aspect can span multiple classes, interfaces, or aspects. Aspects improve separation of concerns by making it possible to cleanly localize cross-cutting design concerns. Different entities will be combined at compile time by invoking a special tool, called a weaver, and they will be weaved into regular Java source files.

Lieberherr et al. in [12] tried to integrate AOP and component-based programming for making aspects more reusable and pluggable. Programmers can use aspectual components to implement higher-level cross-cutting aspects. Aspectual components have `provided` and `expected` interfaces. The expected interfaces consist of an ideal class graph to enable defining one aspect of the system with limited knowledge about the object model and/or other aspects defined by other components. Aspectual components can be deployed into the class graph and composed by third parties by mapping interfaces via explicit connectors.

Because the aspectual components do not need to have hard-wired names of objects that will be connected with the aspectual components, an aspect can be encapsulated into a component and can be reused easily. This is an essential property for applying aspects to EJB, so the concept of aspectual component is used in this research for expressing aspects.

3. Design of AES

Aspect-Oriented EJB Server (AES) is a new EJB server that provides flexibility and extensibility. In AES, aspects can be added and modified with principled interfaces, and various network protocols can be used instead of a fixed RMI [13] protocol. The current implementation supports similar functions, with functions defined in EJB Specification [1] as forms of aspects.

3.1 Definitions of Terms

We now define four terms for clearer discussion.

Definition 1. Meta-level object: A meta-level object is an object that is at the meta-level and implements meta-level functionality. In AES, meta-level is the AES system, so every object that is an element of AES is a meta-level object.

Definition 2. Metaobject: A metaobject is an object that contains information about a baseobject and/or controls the execution of the baseobject. Every metaobject is also a meta-level object, but every meta-level object is not a metaobject. In AES, a metaobject has similar roles as a container in EJB.

Definition 3. Base-level object: A base-level object is an ordinary object that is defined by the Java language specification.

Definition 4. Baseobject: A baseobject is a base-level object that is managed by a metaobject. A baseobject can be composed of base-level objects, but cannot be composed of baseobjects. In AES, a baseobject has similar roles as an enterprise Bean in EJB.

3.2. Classification of Aspects

During this research, we found out need of classifications for aspects used in EJB.

3.2.1. From syntactic point of view

- Orthogonal aspects: An orthogonal aspect has no predefined protocol with a baseobject. Thus, the baseobject does not need any information about the aspect and can be developed independently from the aspect. Container-managed-transaction [1] in EJB, a debugging aspect that displays debugging messages, and a logging aspect that writes the current status information into a storing system are examples of orthogonal aspects.
- Non-orthogonal aspects: A non-orthogonal aspect has a predefined protocol with a baseobject, so a baseobject should have some information about the aspect before developed and has to prepare for being called from the aspect. A developer using the non-orthogonal aspect should consider the aspect while developing an enterprise Bean. An example of this aspect is a stateful session Bean in EJB that requires an enterprise Bean implementing `ejbPassivate` and `ejbActivate` methods that will be called by a container for managing resources. Non-orthogonal aspects may be needed to represent advanced and complex aspects. Whenever possible, however, aspects should be developed as an orthogonal aspect for better portability and reusability.

3.2.2 From a semantic viewpoint

- Basic aspects: A basic aspect is an aspect that has no side effect to a baseobject. We expect that a computation of the basic aspect will not cause a problem for a computation of the next aspect. Therefore, a basic aspect can be ordered at any location in the path of aspect computations. Displaying the data into GUI screen, logging access, security checking, and other many orthogonal aspects can be examples of the basic aspects.
- Special aspects: a special aspect is an aspect that has a side effect to a baseobject after computation of the aspect. A first example of the special aspect is persistence that is related with an identity of a baseobject. If any aspect is computed before a creation or after a destruction of a baseobject, the aspect cannot compute the expected function and raise an exception or cause other problems. Concurrency and transaction are also special aspects.

They provide atomic execution units so a partially inserted aspect with these aspects can cause an unexpected result. A final example of a special aspect is location transparency. In the EJB architecture, a call from a client is a starting point of all other computations even before a computation of a metaobject. Also, some of the routines for location transparency exist in a proxy or a stub at the client side. Therefore, location transparency cannot be controlled by a metaobject in the same way as other aspects and should be treated differently.

3.3 Overall Architecture of AES

In AES, a container is changed into a more generalized metaobject that can support more aspects in a flexible and extensible way. A new source code of a metaobject will be generated when a baseobject is deployed. An aspect exists independently of the metaobject as a Java object using the concepts of aspectual components. Additional aspects can be inserted and updated when they are needed by a *deployer* such as a system administrator of the EJB server. Figure 1 shows the basic elements of the AES architecture.

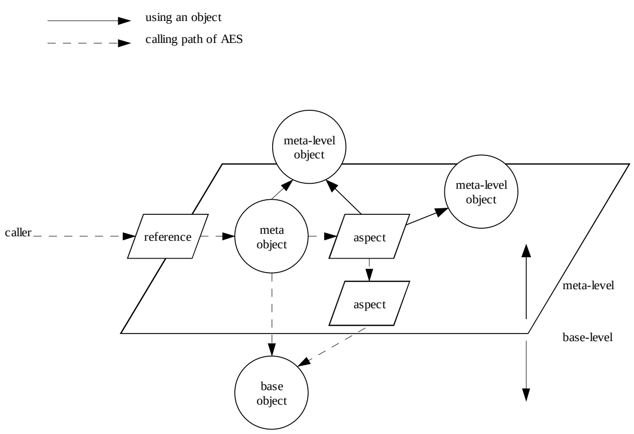


Figure 1. **Overall architecture of AES**

Connections between a baseobject and aspects will be established in the metaobject using information obtained from a deployment descriptor of the baseobject. An updated deployment descriptor in AES can address additional aspects and contain connecting information for a baseobject and aspects. Also, the EJBHome and the EJBObject are changed into reference aspects that can support various network protocols for communication between baseobjects and clients.

3.4 Metaobjects

A metaobject corresponds to a container in EJB but is more generalized. A metaobject is created for a baseobject and has sufficient information about the baseobject. That information, called metadata, is gathered from a deployment descriptor of the baseobject. Also, additional information such as current status of baseobject is updated during run-time. A metaobject has the full control of a baseobject: creating and destroying the baseobject, hooking incoming method calls of the baseobject, and then delegating them to aspects according to the metadata. A metaobject is created as baseobject-specific Java source code, and compiled during deployment-time.

3.5. Aspects

An aspect is implemented with the concepts of aspectual components [12] in AES. An aspect is written in Java and has cross-cutting code for a baseobject such as transaction, persistence, and security. All aspects are used as binary Java class files in AES, not as source files. An aspect has no specific information about a baseobject in its source code, and it will be connected with a baseobject in a metaobject dynamically. A baseobject can use zero or more aspects, and a method in a baseobject can be connected with zero or more aspects.

3.6. Baseobjects

In AES, every Java object can be a baseobject only if it has an interface. The interface does not have to be implemented by the baseobject. The interface represents only what methods of the baseobject will be exposed to clients, and will be used implemented by a metaobject and a proxy object. A baseobject does not have any protocol to follow. However, if a baseobject is connected with a non-orthogonal aspect, then the baseobject should have a set of expected methods. On the other hand, to make an enterprise Bean in EJB, a developer should follow some established rules. The developer should make an enterprise Bean by implementing the `EntityBean` or `SessionBean` interface [1]. Also the enterprise Bean should implement an interface for RMI with predefined signatures of methods.

3.7. Reference Aspects

As described in Section 3.2.2, the separation of location transparency aspect from EJB is difficult. It stems from the architecture of EJB. Therefore, another aspect mechanism is devised for location transparency, called a *reference aspect*. It is a specialized aspect for networking. It also uses the deployment descriptor for gathering information on which reference aspect will be used for a baseobject. While other aspects are called by a metaobject with nested methods, a reference aspect calls a metaobject. The baseobject can have multiple reference aspects. That means a baseobject can be accessed by different network protocols or different policies at a time. A proxy object is created by a reference aspect, and moved to a client system with a serialized form. This proxy object has a same type with the interface for a baseobject described in Section 3.6. A network protocol used for communication between a proxy and a reference aspect can be developed and used with its own network protocol and policy. Usually, a reference aspect can have multiple proxies, but depending on a policy of reference aspect the number of proxies can be restricted. With this architecture, the baseobject can be free from RMI dependent code such as “`throws RMIException`? and a developer can select proper network protocols and policies when a baseobject is deployed.

3.8. Deployment Descriptors

A deployment descriptor has information needed to connect aspects to a baseobject. The information includes (a) the type of reference aspects for a baseobject, and (b) types of the aspect and corresponding expected method in aspects for a method in a baseobject. The syntax of a deployment descriptor is similar to that of aspectual components. An example of a deployment descriptor is illustrated in Figure 2.

```
Store is RMIAspect with {remoteOp = * ;}

Store is IPCheckAspect with
{
  CheckIP = Add*;
  boolean isCorrectIp(String ip)
  {
     if ( ip.equals(? 03.101.93.1”))
           return true;
     else return false;
  }
}
```

Figure 2. **Example deployment descriptor**

The pattern matching on method names is used for mapping of expected operations, so all methods in a class `Store` will be connected with the `remoteOp` method in `RMIAspect`, and Methods starting with dd” in the class `Store` will be connected with the `CheckIP` method in `IPCheckAspect`. Also rogrammatic” mapping can be used as `isCorrectIp` in Figure 2. This programmatic mapping gives more flexibility in using aspects.

4. Implementation of AES

This section describes how the design of AES presented in Section 3 is implemented. Section 4 also describes what problems and issues were and how they were solved in the development process.

4.1 Nested Methods

To support multiple aspects without conflicts between aspects, each aspect should be inserted into another aspect completely. Figure 3 illustrates an example of a method in a metaobject implemented with a cut-and-paste technique.

```
Object foo(arg1, arg2, ?
{
  Object ret;
  code of transaction start
    code of changing a field A from b to b’
      ret = BaseObject.foo(arg1, arg2,? ;
    code of displaying A
  code of transaction commit or rollback
  return ret;
 }
```

Figure 3. **An example of a method in a metaobject implemented with the cut-and-paste**

If the `code of displaying A` was under the `code of transaction commit or rollback`, then an exception or unexpected results in the case of a transaction rollback may occur. Because the value of A is roll-backed from b’ to b but `code of displaying A` does not know about the rollback.

Figure 3 uses a cutting-and-pasting of source code technique for providing aspects in a metaobject. For its efficiency some EJB system including JOnAS and meta-architecture systems select this technique. However, the technique is not recommended from the viewpoint of Software Engineering. Also, it violates the separation of concerns by scattering pieces of code for an aspect over many metaobjects.

Adding before/after methods is generally accepted technique in many meta-architecture systems such as AspectJ, EJBOSS [2], CLOS [16], etc. With this technique, an aspect can exist as an object and utilize all advantages of the object-oriented-programming language. Figure 4 shows the same code of Figure 3 using the before/after technique.

```
Object foo(arg1, arg2,?
{
  Object ret;
     // transaction start
  aspect1.aspectMethod1_before();
     // changing a field A from b to b’
  aspect2.aspectMethod2_before();
  ret = BaseObject.foo(arg1, arg2, ? ;
     // displaying A
  aspect2.aspectMethod2_after();
     // transaction commit or roll-back
  aspect1.aspectMethod1_after();
  return ret;
}
```

Figure 4. **An example of a method in a metaobject implemented with the before/after methods**

Using this technique is faster than nested methods, but it has problems to provide full control of an execution. In nested methods, parameters, return value, and exceptions are propagated smoothly between aspects and a baseobject. However, using before/after methods, additional code is required to deliver parameters, return value, and exceptions. An exception is especially hard to be controlled with before/after methods. For example, in Figure 4, if an exception is raised in the `aspect2.aspectMethod2_before` method, then the `aspect1.aspectMethod1_before` method should know about the exception. However, when the exception takes place, already a context of `aspect1.aspectMethod1_before` method is over and the exception cannot be passed to the method, so the exception should be passed to the object `aspect2` or the method `aspect1.aspectMethod1_after`. In both cases, they have difficulties to make the same context of `aspect1.aspectMethdo1_before`. Because local variables cannot be preserved after exiting a method and preserving the context as fields of `aspect1` makes sharing problems when multiple clients or threads access the aspects. The piece of code in Figure 3 can be rewritten using nested methods as follows:

```
Object foo( arg1, arg2, … )
{
  Method m1 = BaseObject.getClass().
                  getMethod(□oo? ;
  Method m2 = aspect2.getClass().
                  getMethod(□spectMethod2? ;
  Method m3 = aspect1.getClass().
                  getMethod(□spectMethod1? ;
  NestedMethod nm = new NestedMethod(
                        this, baseobject, m1);
  nm.addMethod(aspect2, m2);
  nm.addMethod(aspect1, m3);
  Object[] args ={arg1, arg2, ? ; // line 1
  return nm.invoke(args);         // line 2
}
```

Figure 5. **Rewritten code with nested methods**

A nested method object has a list of Method objects. In figure 5, when the `invoke` method at `line2` is called, the nested method object `nm` calls the last added method, `m3`. Then, the `m3` method calls `m2` when the nested object is invoked one more time in the body of `m3` method. The nested method object is passed as an argument to the methods of the aspects such as `m2` and `m3`. With the same way, the `m2` method calls the `m1` method of the baseobject. To decrease overheads of reflective calls, every line of code except `line1` and `line2` in Figure 5 is moved to an `init` method and reused for each method call. The `init` method is executed once when an instance of the metaobject is instantiated and never called again. Figure 6 shows a calling path of code in Figure 5.

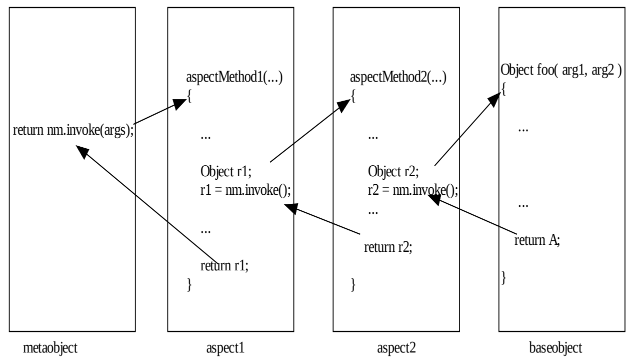


Figure 6. **A calling path of nested methods**

In this calling path, all methods are executed in a context of a prior executed method, so propagation of arguments, return value, and exceptions are easier and more natural than the before/after method technique. Also, local variables in each method of aspects are preserved during execution of a next nested method, so multiple clients or threads can access a metaobject and an aspect simultaneously without interference.

4.2 MetaObjects

In AES, we assume that (a) the changes of aspects being used by a baseobject are not frequent, and (b) the cost of making a call from a metaobject to a baseobject is critical as a middleware, although the cost of deployment is not critical. Therefore, to reduce overhead of reflective calls, source code of a metaobject is created in deployment-time and compiled on-the-fly. A one-to-one mapping between a baseobject and a metaobject exists, so when new instance of a baseobject is created, new instance of the metaobject also created.

4.3 Aspects

To write code for an aspect, a developer of the aspect follows two rules. These rules are (a) every aspect should inherit the `AspectualComponent` class, and (b) every method that will be called by a metaobject should have a fixed signature as follows:

```
public Object methodName(MetaObject caller, ClientMetaData cm, Object target, Object[] args, NestedMethod m) throws Exception
```

- A type of return-value is `java.lang.Object`. Casting to the type of the return-value for a baseobject is processed by the metaobject.
- `MetaObject caller` is a metaobject of the baseobject.
- `ClientMetaData cm` has additional information that includes client-specific data such as IP address of the client, name of the reference aspect, unique handle of the metaobject, an identity of client, etc.
- `Object target` is a baseobject.
- `Object[] args` is an array for a method of the baseobject.
- `NestedMethod m` has methods of aspects and a method of the baseobject.

The aspects can use all functionalities of Java language such as dynamic class loading, inheritance, encapsulation, code-mobility, etc. An instance of an aspect is created in the instantiation stage of a metaobject. This instance is known only to the metaobject, and exists as a field of the metaobject. Therefore, if the metaobject is not used more, then the instance will be finalized with the metaobject together.

During implementation of AES and a testing application, many aspects are developed as follows:

- `AspectPersistence` provides persistence a to baseobject using JDBC [5].
- `AspectSync` provides object-level synchronization.
- `AspectTx` provides simple transaction with JTA [8].
- `AspectSecurity` checks an object of `java.security.Identity` [7] delivered from a client to a metaobject before executing a method.
- `AspectLog` stores access information from clients with thread and time information.
- `AspectIPTrace` is designed to monitor clients. The aspect shows periodical statistics of access information with client IP addresses.
- `AspectMax` checks whether a value of an argument exceeds its limit or not.
- `AspectDistributedTx` supports distributed transaction. It receives additional information of a distributed transaction from a client and returns that to the client with `TxRMIAspect`. Figure 7 shows the code of `AspectDistributedTx`.

```
public class AspectDistributedTx extends
AspectualComponent implements Synchronization{
  …
  public AspectDistributedTx()
  { tm_ = Current.getCurrent(); }
  public Object Tx_Required(MetaObject caller,
ClientMetaData cm, Object target, Object[]
args, NestedMethod m) throws Exception{
    …
    currentTx = tm_.getTransaction();
    if ( currentTx == null ) {
      tm_.begin();
      currentTx = tm_.getTransaction();
      must_commit = true;  }
    currentTx.registerSynchronization(this);
    synchronized (this){
      try{ ret  =  m.invoke(args, cm); }
      catch (Exception e){
        flag_exception = true;
        throw e;}
      finally{
        if ( must_commit ){
           if (flag_exception) tm_.rollback();
           else  tm_.commit();
        else {
           if (flag_exception) {
            currentTx.setRollbackOnly();
            throw new
    TransactionRolledbackException("? );}}
      } // end of finally } // end of sync
    return ret;}}
```

Figure 7. **AspectDistributedTx**

4.4. Reference Aspects

As explained in Section 3.2.2, location transparency is treated differently and reimplemented as a reference aspect. Figure 8 shows how a reference aspect is created and works.

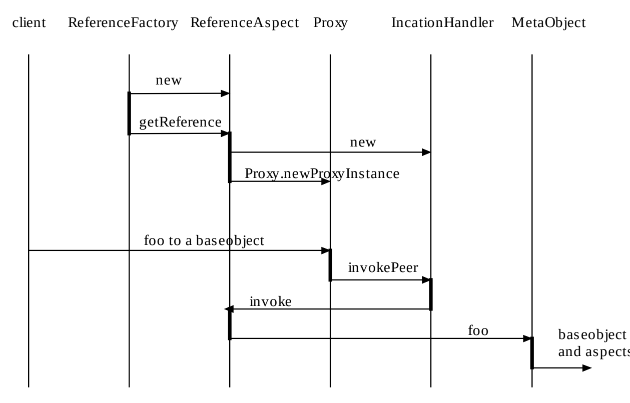


Figure 8. **Creating and working path of a reference aspect**

The `ReferenceAspect` is an abstract class, so subclasses implement the `getReference` abstract method. In the `getReference` method, a subclass makes a proxy object with an invocation handler. The invocation handler is specific to a subclass of `ReferenceAspect`, and these are used together. Each subclass of `ReferenceAspect` can use its constructor to add specific code for networking such as registering itself as a `UnicastRemoteObject` [13] for using RMI. The returned proxy object from the `getReference` method will be registered to JNDI [6] with a unique name. So when a client wants to use a baseobject with the name, then the proxy will be delivered to the client. A proxy object is implemented with a dynamic proxy object [14] in JDK1.3 that implements a list of interfaces specified at runtime. An invocation handler is created in the `getReference` method and used in a proxy object. Every invocation handler is be a subclass of an `AESInvocationHandler` that has an abstract method, `invokePeer`. When a client calls a method in a proxy object, the call will be delegated to an `invoke` method in an invocation handler. After basic work such as making a `ClientMetaData`, the `invoke` method will call the `invokePeer` method. Then, the `invokePeer` method usually calls a reference aspect with a specific network protocol. Currently, `RMIAspect`, `RMIAspect2`, `LocalAspect`, and `TxRMIAspect` are developed as follows:

- `RMIAspect` provides a location transparency aspect. It uses RMI for remote calls, and makes a `UnicastRemoteObject` per an instance of metaobject.
- `RMIAspect2` has a same function with `RMIAspect` but makes a `UnicastRemoteObject` per a type of metaobject.
- `LocalAspect` is used for local-calls in a JVM. The invocation handler calls its reference aspect directly. With this aspect, baseobjects can be used from clients or other baseobjects without a detour round RMI.
- `TxRMIAspect` has a same functionality of `RMIAspect` with supporting distributed transaction. When a client calls a method, the `invokePeer` method of the `TxRMIInvocationHandler` gets current transaction information and puts the information into arguments array. Then, the `invoke` method of `TxRMIAspect` extracts the transaction information and sends that to a metaobject. After finishing all needed execution in a metaobject, the `TxRMIAspect` returns the transaction information to the `TxRMIInvocationHandler` with the same way.

A reference aspect using another network protocol or different policies can be developed with (a) the implementation of constructor and the `getReference` method in the `ReferenceAspect` subclass, and (b) the implementation of the constructor and the `invokePeer` method in their `AESInvocationHandler` subclass. Because a baseobject does not have any reference aspect specific code, it can use any reference aspect by just selecting what reference aspect it wants to use in a deployment descriptor.

We expected other reference aspects such as reference aspect for streaming media data, reference aspect using CORBA, or reference aspect using UDP can be developed easily.

5. Performance

Because performance is usually gained by sacrificing some flexibility and extensibility in a system, performance issues are important variables to measure AES flexibility and extensibility. We tested three kinds of experiments with a testing application that provides basic functions for e-commerce. One is a performance comparison with other EJB servers, and another is measuring costs for supporting aspects. Finally, we compared cost of nested methods with other techniques.

5.1. Comparison with JOnAS

To test performance of AES and other EJB servers, we produced similar environments as much as possible. First, we selected the JOnAS for comparing with AES. Because its architecture is designed for performance not for flexibility, we would get practical outputs by comparing with JOnAS. We selected same database, InstantDB [3], and the same database options. AES uses JTA implementation of JOnAS, so the transaction routines would not add undesirable differences. For JOnAS, we used an entity bean with options such as container-managed-persistence, container-managed-transaction, and TX_REQUIRED. These options make similar conditions with aspects used for testing in AES. Finally, we wrote two testing applications for AES and JOnAS. The two applications have few syntactic differences in source code and execute semantically same functions. To simulate environments of multiple clients, the application creates multiple instances with threads. All our measurements were collected using the JDK1.3 RC2 Hotspot Client VM on a lightly loaded 1-CPU 400MHz PentiumII system with 128 megabytes of RAM and Windows 4.0 Workstation. For networking, we used loop-back. The following table shows the result of this experiment.

Table 1. **Results of comparison with JOnAS**

	# executions	100	500	1000
1 client	RMIAspect	3773.75	18542.40	36047.00
	RMIAspect2	4237.80	19831.00	39894.50
	JOnAS	7218.33	35891.00	75664.00
10 clients	RMIAspect	31585.90	171993.90	351010.90
	RMIAspect2	36317.30	176192.30	358771.80
	JOnAS	54124.75	298398.50	596891.00
50 clients	RMIAspect	167436.90	1144601.90	
	RMIAspect2	175691.90	1184765.04	
	JOnAS	166406.00		

Each value of fields in the table represents milliseconds for executing loops. A loop is composed of a method call to a baseobject in AES and an enterprise Bean in JOnAS, and this call sets a field with a new value and writes the change into a database. In the case of 10 and 50 clients, the values are an average of clients' results. JOnAS uses a `UnicastRemoteObject` per a type of an enterprise Bean like RMIAspect2. Because of a server exception, we could not get a value for JOnAS in the case of 50 clients and 500 time loops. Also in the case of 50 clients and 1000 time loops, we could not get values for all cases with the same reason. We expect that is caused from the transaction module that is used in both of AES and JOnAS. We made an assumption for getting a response time in this experiments as follows:

$$\textit{Response time}_{\textit{a method call}} = \frac{\textit{required time for executing loops}}{\textit{number of executions}}$$

With this assumption, following graphs can be developed in Figure 9 for the response time.

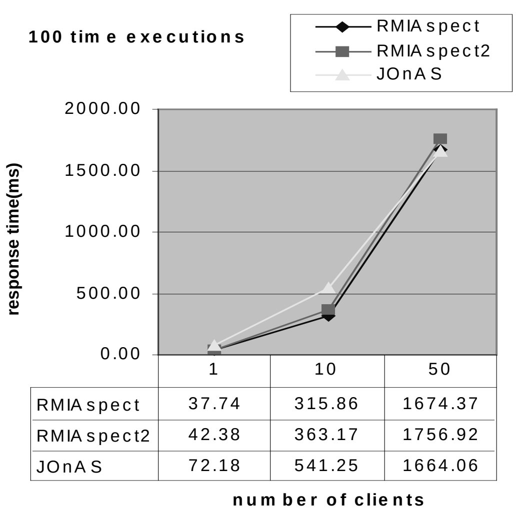


Figure 9. **Response time comparing with JOnAS**

With these results, we can discover the response time has no noticeable difference between RMIAspect, RMIAspect2, and JOnAS in the both 100 time and 500 time execution cases. Also it shows that AES is faster than JOnAS in some cases. The result, AES is faster or not slower than JOnAS, can be analyzed with those reasons: (a) reflective calls in nested method are well optimized, (b) JOnAS does additional work to support large-scale requests of clients. The first reason will be examined in the next section in the details. The second reason can be proved with source code of JOnAS. JOnAS manages a pool of objects to reuse the objects without destruction. The management can be overhead in the early stage, but can provide more efficiency in the case of more clients and more loops as illustrated in Figure 9.

5.2. Cost of Aspects

We measured the architectural cost required to provide flexibility and extensibility for EJB. Figure 10 shows

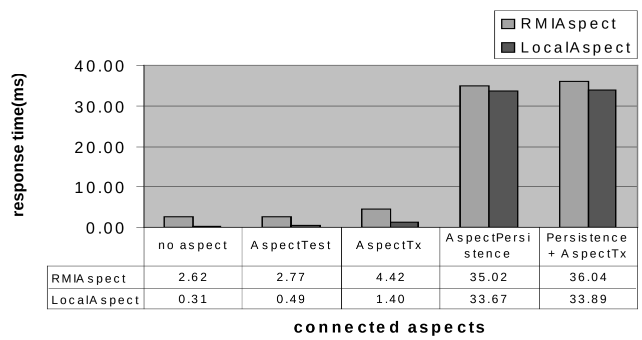
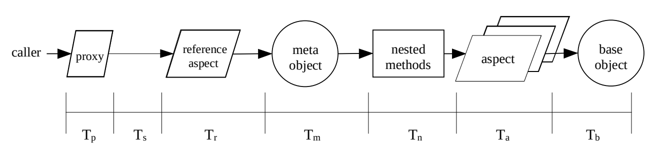


Figure 10. **Response times of using aspects**

response times of using aspects.

This is a result of tests with same conditions described in Section 5.1 in the case of 1 client and 1000 time loops. A response time can be analyzed with this equation as follows:



$$\textit{Response time (RT)} = T_p + T_s + T_r + T_m + T_n + T_a + T_b$$

- T_p: time consumed in a proxy for making an instance of `ClientMetaData`, marshalling additional parameter, etc.
- T_s: time consumed in network protocol and network protocol routines such as propagation delay and an RMI skeleton.
- T_r: time consumed in a reference aspect for finding metaobject, unmarshalling additional parameter, etc.

- T_m: time consumed in a metaobject for making an argument array and casting a return type.
- T_n: time consumed in a nested method for reflective calls.
- T_a: time consumed in aspects.
- T_b: time consumed in a method of a baseobject.

In this equation, `RMIAspect` and `LocalAspect` have differences at only T_p, T_s. Definitely T_r, and T_s are zero In `LocalAspect`. If any aspect is not used, T_n and T_a also are zero, therefore

$$RT_{LocalAspect} = 0.31 = T_p + T_r + T_m + T_b$$
$$RT_{RMIAspect} = 2.62 = T_p + T_s + T_r + T_m + T_b$$

In T_p and T_r, there is no noticeable difference in their source code between these two reference aspects, so we assumed T_p and T_r are same between `RMIAspect` and `LocalAspect`. Then, those equations can be written as follows:

$$T_s = 2.62 - 0.31 = 2.31 \text{ ms}$$

We can notice similar differences such as 3.05, 2.28, 1,35 and 2.51 for each column in Figure 10.

Also T_n and T_a for `AspectTest` can be calculated as follows:

$$T_{aspect} = T_n + T_{a\ AspectTest}$$
$$= RT_{AspectTest} - RT_{no\ aspect}$$
$= 2.77 - 2.62 = 0.15$ for `RMIAspect`
$= 0.49 - 0.31 = 0.18$ for `LocalAspect`

The `AspectTest` has no additional code in the method and just executes a method of a baseobject so $T_{a\ AspectTest}$ can be regarded as the cost of applying one aspect. T_{aspect} is what we want to get in this experiment, and required cost for adding an aspect. We can notice the portions of T_{aspect} within total response times are from 0.41% to 5.41 % when using the `RMIAspect` and from 0.53% to 36.7% when using the `LocalAspect`.

5.3 Cost of the Nested Methods

To measure cost of nested methods, we made one more experiment with the cut-and-paste technique and the before/after methods technique as described in Section 4.1. We used source code of a metaobject created by AES with `AspectTest`. This aspect does not have any code inside of methods. For testing of the cut-and-paste technique, we modified the metaobject code and assumed additional code for aspects will be cut-and-pasted in a method as in Figure 3. However, we did not put any code except a direct method call of a baseobject because the `AspectTest` has no code in its methods. For testing of the before/after technique, similar code to the Figure 4 was used. The before/after methods had no code like other techniques. All methods were invoked without reflections.

We tested with same conditions as in Section 5.1, but we made another small program to decrease other variables such as T_p, T_s, *and Tr* in Section 5.2. This program makes an instance of the metaobject and calls a method of the metaobject 1000 times directly. Results are shown in Table 2.

Table 2. **Testing results of meta-architecture techniques**

Cut-and-paste	Before/after	Nested methods
0.013 ms	0.015 ms	0.109 ms

By combining values of Table 2 and the result in Section 5.2, we calculated how many differences would happen in performance when we use the cut-and-paste or the before/after methods techniques instead of nested methods. Table 3 shows this result.

Table 3. **Response time comparing between meta-architecture mechanisms**

		nested methods	cut & paste	%	before /after	%
AspectTx +	RMI	36.04	35.94	100.27	35.95	100.26
AspectPersistence	Local	33.89	33.79	100.29	33.80	100.28
AspectTest	RMI	2.77	2.67	103.61	2.68	103.50
	Local	0.49	0.39	124.60	0.39	123.72
AspectTx	RMI	4.42	4.33	102.23	4.33	102.16
	Local	1.40	1.30	107.40	1.30	107.17

Values in the first column are same with the values of Figure 10 and represent response times using nested methods. A value in the second column is calculated with this equation:

RT_{new} = a value of first column
– a value$_{\text{nested methods}}$ in Table 2
+ a value$_{\text{cut-and-paste}}$ in Table 2

The value of nested methods in Table 2 is corresponding to $T_m + T_n + T_a + T_b$ in Section 5.2. In Section 5.2, we obtained $T_{aspect} = 0.15$ for `RMIAspect` and 0.18 for `LocalAspect` and this value does not include T_m and T_b, but those values are bigger than the value in Table 4.2 for nested methods. We analyze this difference is caused from systematic error affecting all results in Table 4.2. Reasons of this systematic error can be different states of systems such as CPU, memory, JVM, etc. Therefore we expect errors will be cancelled out during the calculation for an RT_{new}. A value in the third column represents a percentage of an $RT_{nested\ methods}$ per an RT_{new}. In fourth and fifth column, the same equation is used with 0.015 instead of 0.013 in the second column. Note that at the most, using a nested method is slower 1.0361 times than using cutting-and-pasting in the case of `RMIAspect`, and 1.246 times slower in the case of `LocalAspect`.

6. Conclusions

The goals of the thesis were to examine the capability of an EJB server as an AOP environment, and to develop an EJB server that is able to provide additional and modified aspects. These goals were accomplished and were demonstrated with a prototype implementation called AES that is more generalized with a meta-architecture without serious performance degradation. AES separates a baseobject from aspects such as persistence, transaction, security, and even location transparency. Also, other user defined aspects can be applied with a combination of other aspects.

7. References

[1] Sun Microsystems, Inc., nterprise JavaBeans™ (EJB) specification 1.1? Available from http://www.javasoft.com/products/ejb.

[2] An EJB Open Source Server, Available from http://www.ejboss.org.

[3] InstantDB, Available from http://instantdb.enhydra.org/.

[4] Java™ Open Application Server, Available from http://www.bullsoft.com/ejb.

[5] Sun Microsystems, Inc., DBC 2.1 Core API specification? Available from http://www.javasoft.com/products/jdbc.

[6] Sun Microsystems, Inc., NDI 1.2 documentation? Available from http://www.javasoft.com/products/jndi.

[7] Sun Microsystems, Inc., ava™ Security Architecture? Available from http://www.javasoft.com/security/.

[8] Sun Microsystems, Inc., "Java Transaction API (JTA) 1.0.1 Specification? Available from http://www.javasoft.com/products/jta.

[9] Gregor Kiczales and Cristina Videira Lopes, "Aspect-Oriented Programming? In *Proceedings of the ECOOP97*, Springer-Verlag LNCS 1241, 1997.

[10] T. Ledoux, penCorba: a Reflective Open Broker? In *Proceedings of the Reflection'99*, LNCS, vol. 1616, Springer Verlag, 1999.

[11] Cristina Videira Lopes and Gregor Kiczales, ecent Developments in AspectJ , In *ECOOP'98 Workshop Reader*, Springer-Verlag LNCS 1543, 1998.

[12] Karl Lieberherr, David Lorenz, and Mira Mezini, rogramming with Aspectual Components? Technical Report, Northeastern University, 1999.

[13] Sun Microsystems, Inc., ava™ Remote Method Invocation (RMI)? Available from http://www.javasoft.com/products/jdk/rmi/index.html.

[14] Sun Microsystems, Inc., ynamic Proxy Classes? Available from http://java.sun.com/products/jdk/1.3/docs/guide/reflection/proxy.html.

[15] Junichi Suzuki and Yoshikazu Yamamoto, penWebServer: an Adaptive Web Server Using Software Patterns? *design patterns of IEEE Communications Magazine*, 1998.

[16] Gregor Kiczales, Jim des Rivi?es, and Daniel G. Bobrow, "*The Art of the Metaobject Protocol?* MIT Press,1991.