Applying Enterprise JavaBeans – Architecture & Design

Adrian M Colyer Andrew Clement Steve Longhurst

IBM UK Limited
Hursley Park
Winchester
England, SO21 2JN

Abstract. The Enterprise JavaBeans™ specification[1] represents a collaborative effort led by Sun Microsystems™ with contributions from many vendors including IBM®. It defines a component architecture for distributed enterprise-level applications. This differs from ordinary object-oriented development in that it addresses *components* rather than objects, and in that it addresses *distributed* rather than local computation.

Architecture and design of a distributed component-based application raises different challenges and design points than for a non-distributed object-based application.

Now that products such as IBM's WebSphere™ Application Server [2] that support the Enterprise JavaBeans specification have been in the marketplace for a while we can begin to see some of the common pitfalls and design patterns that are emerging.

This paper examines the role of enterprise beans in enterprise Java™ based e-commerce applications, and then goes on to look at design guidelines to improve the performance and reliability of enterprise bean based applications.

We conclude with a look at some of the challenges for researchers and vendors as the Enterprise JavaBeans specification moves on to its 2.0 release and beyond.

Keywords. Enterprise JavaBeans, distributed application design, component models, e-commerce.

1 INTRODUCTION

Application servers have been around for a long time in various guises. For example, IBM's CICS®[3] is an application server that provides industrial strength, online transaction management for mission critical applications – and it has been in the marketplace for over 30 years!

The term "application server" can be loosely defined as an execution environment and set of runtime services made available to applications written to take advantage of those services.

Recently we have seen the emergence of the "Web Application Server" – an environment providing for example HTTP request serving, support for servlets[4] and Java ServerPages™[5], connection management and related facilities.

For enterprise applications, the term *component transaction monitor* was introduced by Anne Thomas of the Patricia Seybold Group. A component transaction monitor (also sometimes called a component transaction server) is a sophisticated distributed object application server combining traditional TP monitor and distributed object technology.

Web Application Servers and Component Transaction Monitors work in tandem to deliver enterprise level solutions for e-commerce. This relationship will be explored later in the paper.

The applications executed in a component transaction monitor are typically composed of *components*. A component can be thought of as an application building block that adheres to the conventions of a given component model. By following a component model, components can inter-operate, take advantage of services defined as part of the component model (for example, lifecycle and transaction management), and be deployed on servers that comply to the component model.

The piece of a component transaction monitor that provides components deployed within it with an environment adhering to a given component model is known as a "container."

There are currently two widely deployed component models for component transaction monitors. The Enterprise JavaBeans (EJB) component model is a cross-industry collaboration owned and managed by Sun through the Java Community Process (JCP). A wide range of products from a number of vendors support it. The COM+ component model is developed by Microsoft™ and supported by the Microsoft Transaction Server (MTS)[6]. Comparisons of EJB and COM+/MTS can be found at [7],[8].

A third component model, the CORBA Component Model (CCM) is under development by the OMG [9].

2 ENTERPRISE JAVABEANS

The EJB component model provides a cross-platform and cross-vendor standard for components written in the Java programming language. These components are known as "enterprise beans."

The model allows a developer to focus solely on writing the business logic of an enterprise bean. The way that the bean interacts with its environment - for transaction services, security services, persistence management and so on is separated from the business logic and instead specified via a "deployment descriptor."

Deployment descriptor information is typically captured by tooling that is provided by an EJB vendor in conjunction with their server runtime. For example, IBM allows production of enterprise beans and specification of deployment attributes from within the VisualAge® for Java[10] development environment. Deployment data can be changed independently of the business logic, and it is the responsibility of the EJB container to honor the contract laid out in the deployment descriptor.

For example, the transaction attribute deployment descriptor allows specification of

the transaction management to be applied upon invocation of a business method of an enterprise bean. The available transaction attributes in version 1.1 of the specification are listed in table 1.

Table 1. Enterprise Bean Transaction Attributes**.**

Attribute	Meaning
NotSupported	Bean method is invoked outside of a transaction.
Supports	Bean method invoked within existing transaction if present.
Required	Bean method invoked within existing transaction if present, otherwise a new transaction is started.
RequiresNew	Bean method invoked within a new transaction.
Mandatory	Bean method must be invoked as part of an previously existing transaction.
Never	Bean method must never be invoked as part of a transaction.
BeanManaged	Bean manages its own transactions.

The separation of concerns afforded by the use of deployment descriptors is a big step forward from the pre-component world where a developer had to learn and master the various APIs needed to interact with runtime services (such as the CORBA runtime services) and explicitly code this into their application.

An EJB Container honors the contract laid out in the deployment descriptor via *interposition*. A developer writing an enterprise bean defines its remote interface - the set of business methods that the bean will service, and its home interface - the means to create, find and remove instances of the bean. He or she then writes the business logic in an enterprise bean class that implements the business methods, creation and finder methods following the pattern laid out in the EJB component model.

When an enterprise bean is deployed into an EJB Container the deployment tooling provided by the container vendor generates additional classes (a *Home* class and a *Remote* or *EJBObject* class) that implement the home and remote interfaces defined by the bean developer.

A client of an enterprise bean does not have direct access to the enterprise bean, but is forced to access it through the generated classes. The generated classes work in conjunction with the container to provide pre and post-invoke processing that wrappers client calls to the bean and interposes the necessary transaction, security and persistence management.

This is illustrated in figure 1. A Java client uses RMI/IIOP (Java's Remote Method Invocation programming model flowing over the OMG's IIOP protocol for internet based ORB interoperability) to communicate with an EJB Server.

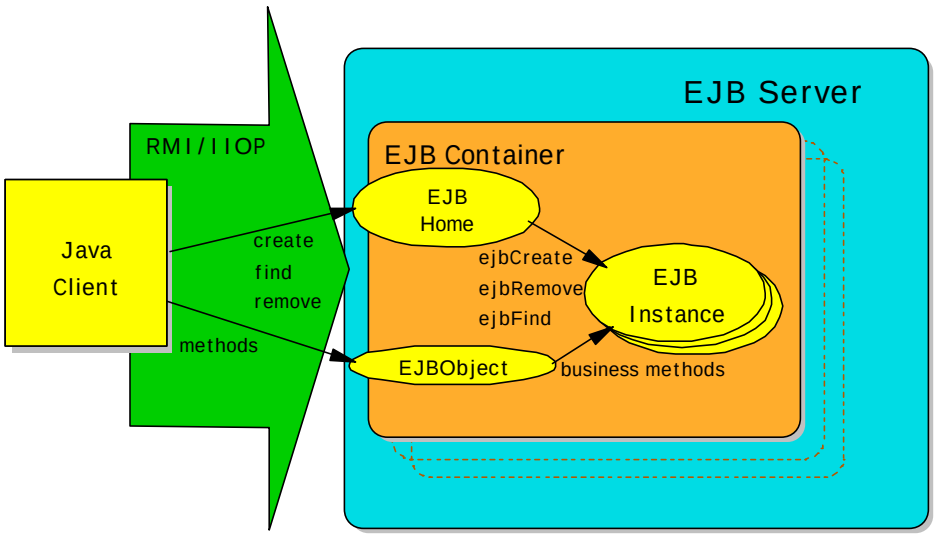


Fig. 1. Container Interposition on Enterprise Bean Method Invocation

When the client wants to create, find or remove instances of the enterprise bean it first obtains a reference to the bean's home object by looking up a well-known name via the naming service. In the EJB component model, the interface to the naming service is provided by the JNDI™ API[11].

The home object reference returned from the name lookup is to an instance of the Home class generated by the deployment tools when the enterprise bean was deployed into the EJB server. This class implements the home interface defined by the bean developer. Typically it will perform some environment set-up (for example, verifying access to run the target method), and then delegate the call to methods provided on the enterprise bean class provided by the bean developer. When the enterprise bean method has completed, the generated Home class may perform some post-invoke processing (for example, committing a transaction that it started) before returning to the client.

When an enterprise bean is created or found via its home, a reference to the remote (or EJBObject) class is returned to the client. The EJBObject implements the remote interface defined by the bean developer. When the client invokes the business methods defined in the remote interface the EJBObject performs pre-invoke processing (such as starting a transaction if one is required). It then delegates the method call to the enterprise bean class written by the developer (which contains the business logic), and then performs any post-invoke processing required before returning to the caller.

Thus the EJB container interposes on calls made from the client to the enterprise bean, which is the mechanism by which it is able to honor the contract specified in the enterprise bean's deployment descriptor.

2.1 Types of Enterprise Bean

The EJB component model defines two different kinds of enterprise beans: session beans and entity beans.

Session beans are for use by a single client and provide a pseudo-conversational interaction style representing a "session" or sequence of interactions between a client and the EJB server. A *stateful* session bean contains state particular to its client, whereas a *stateless* session bean contains no client-specific state. Stateless session beans are the highest performing class of enterprise bean since work can easily be balanced across a pool of equivalent bean instances. A session bean is non-recoverable in the event of a failure.

In contrast, entity beans are recoverable and are shared by multiple clients. They provide an object rendering of application data. Entity beans are indexed by a primary key (created, found, removed), and are somewhat analogous to rows in a database table. A bean-managed persistence (BMP) entity bean manages its own state, whereas a container-managed persistence (CMP) entity bean has its persistent state managed by the container.

3 THE ROLE OF ENTERPRISE BEANS IN E-COMMERCE APPLICATIONS

To best understand where enterprise beans fit in an e-commerce architecture we need to explore briefly the history of dynamic web-based applications and the motivations for each shift in technology as it occurred.

The first dynamic web sites used something called CGI (Common Gateway Interface) to build dynamic web pages. Using CGI, each request was serviced in a separate process kicked off by the web server. The CGI programs themselves performed three basic tasks – parsing of the HTTP request headers, business logic to process the request, and formatting of the response page.

Launching a new process to service each incoming request did not scale well as web sites started to handle higher volumes of traffic. Web server vendors started to introduce server-specific APIs that enabled requests to be serviced on a thread of an existing web server process. For example, Netscape introduced the NSAPI, and Microsoft the ISAPI. This enabled faster processing of requests, but at the cost of portability across web servers and across platforms. Programs written to work with these server-specific APIs still mixed request parsing, business logic and presentation.

Sun introduced servlets [4], which retained the performance advantages of the server specific APIs by running requests inside a thread of a web server process. In addition, because the servlet interface was defined as a standard, servlets could be run on any web server supporting that interface. Since servlets are written in the Java programming language, they can also be run on any platform.

The next technology to be introduced was JavaServer Pages or JSPs [5]. Like the

technologies that had gone before, servlets mixed business logic with presentation logic. That confused the role of developer and web page designer – which are differing areas of expertise - and made maintenance problematic. Using JSPs and servlets in combination allowed program logic to be placed in the servlet, and presentation in one or more JSPs.

Adding JavaBeans™ [12] into the mix allowed separation into a model-view-controller design where servlets (the controller) parse incoming HTTP requests and delegate to JavaBeans (the model) which perform the business logic. After the JavaBeans have done their work, the servlet (controller) then passes control to a JavaServer Page (the view) which builds the output page by extracting data values from the JavaBeans.

In a three-tier e-commerce solution, clients (typically a web browser) communicate via HTTP with a web application server that combines HTTP request serving with support for servlets and JSPs. The JavaBeans implementing the business logic use JDBC™ [13] to communicate with a third-tier data source (sometimes referred to as an Enterprise Information Server or EIS) at the back end.

As companies grow more sophisticated in their exploitation of the web and increase the integration of their back-end systems with web front-ends the business logic behind web applications grows more complex. Particularly there is a need for transactions to support co-ordination of updates across multiple data sources to ensure data integrity. Adding transaction management, security, persistence and other such services to create a true enterprise class application is the role of Enterprise JavaBeans.

The Enterprise JavaBeans solution adds a component transaction monitor at tier three. The JavaBeans at the web application server act as a local proxy for enterprise beans running within the component transaction monitor.

A typical deployment scenario for this configuration is illustrated in figure 2. The browser (or other thin client such as a pervasive computing device) resides on the internet. A firewall protects the web application server from anything other than incoming HTTP requests, which are processed by the web application server within what is known as the “Demilitarized Zone” or DMZ. Connections from the web application server to the EJB server (component transaction monitor) pass through another internal firewall (which is typically configured to only allow connections from the web application server host). Inside the second firewall is the company’s secure intranet where the enterprise beans running within the EJB Server will typically interact with one or more back end databases.

We now turn our attention to the architecture and design of the enterprise bean components running within the component transaction monitor.

4 DESIGN GUIDELINES

Enterprise JavaBeans defines a distributed component model.

A *distributed* model increases the cost of communication exchanges between the client application and the EJB server, and also between enterprise beans distributed across multiple servers on multiple hosts.

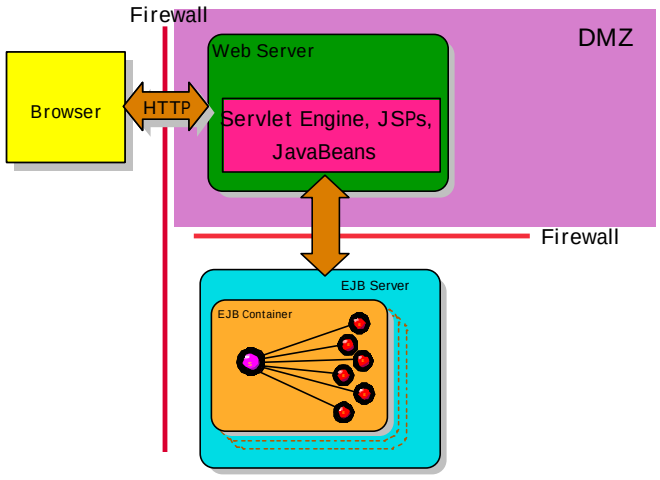


Fig. 2. A typical enterprise beans based e-commerce deployment.

A *component* model can also increase the cost of inter-component communication due to the overheads of interposition by the container. Over and above the considerations of conventional object-oriented design, the EJB application designer must also consider:

- The appropriate use of distribution, and granularity of communication
- The granularity of components (many finer-grained components, or fewer, coarser-grained components)
- Transaction boundaries and the use of the transaction attribute deployment parameter.
- Persistence mapping between entity beans and the underlying persistent data source(s).

For this section we will use as an example a basic application for capturing orders from customers. In a deliberate simplification there are four entity beans in our initial system as illustrated in figure 3.

The *Customer* entity bean contains information about a customer's name, address and order history. The *Part* entity bean contains information about available parts in the company's catalogue. The *Order* entity bean represents an order made by a customer. Orders consist of a number of items, each *OrderItem* for a given quantity of a part in

the catalogue.

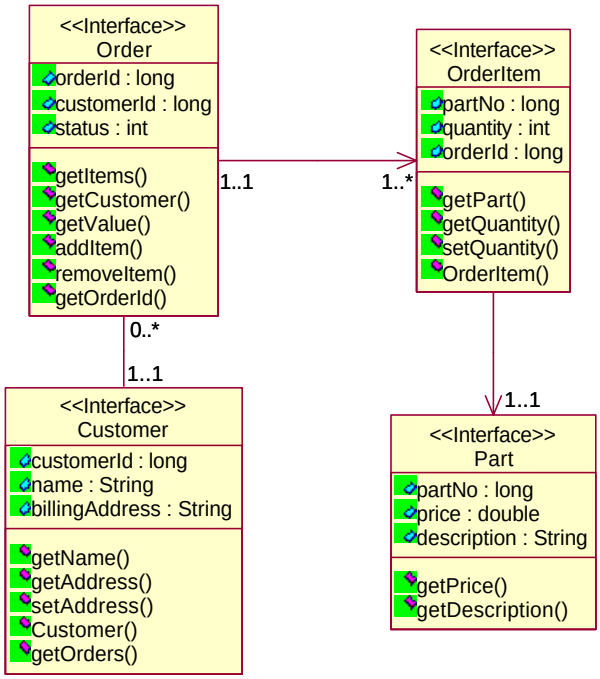


Fig. 3. First pass at entity bean remote interfaces

4.1 Façade / Proxy

Consider a client application that prints a shipping report for a given order identified by an orderId. The shipping report contains the customer's name and billing address, an itemized list of the parts in the order, and the total order value.

With our system as it currently stands, our client application would have to:

1. Lookup the *Order* bean home object in the naming service
2. Find the *Order* bean representing the given order, using the *findByPrimaryKey* method defined on the home interface.
3. Obtain a reference to the *Customer* bean, and from there the customer's name and address.
4. Ask the *Order* for its *OrderItems*.
5. For each *OrderItem*, get the corresponding *Part*, part details and quantity.

For an order containing only three items, this requires as many as eighteen remote communications between the client and the server. If there were thirty items in the order, it would require a staggering one hundred and twenty-six remote communications. This approach performs poorly and does not scale well.

Stretching the example slightly, suppose that multiple clients may operate on the order in parallel. The client requires a self-consistent shipping report (the order value, and the sum of the prices of the ordered items must agree for example). To guarantee this with the design as it stands the client must bracket all of the calls within a single transaction. The transactional locks will be held for an eternity (in transaction processing terms) whilst all the remote communications occur, significantly degrading system performance as the workload is scaled.

A sample client program illustrating this solution is shown in listing "A" at the end of this paper.

To overcome some of these problems we introduce a session bean into the equation. The session bean will reside on the EJB server with the entity beans, and act on the client's behalf. Let's call it the *Report* bean. This bean defines a method *getShippingReport*, which takes as input an order id, and returns a formatted report as a string.

This technique is varying known as the *proxy* pattern (because the session bean acts as a local proxy for the client), the *façade* pattern (because the session bean provides a façade in front of a number of entity beans), or by the name "Session fronts Entity."

For the client application to obtain a shipping report for a given order id now, it requires only three remote communications: one to obtain a reference to the *Report* home, one to create a *Report* instance, and one to call the *getShippingReport* method. The number of remote calls stays constant regardless of the number of items in the order. The implementation of the *getShippingReport* method in the session bean interacts locally with the entity beans to build the shipping report as before.

We can easily bracket the whole thing inside a transaction by marking the *getShippingReport* method with the TX_REQUIRED transaction attribute. This is simpler than creating a user transaction in the client as we had to do previously, and has the advantage that the transaction stays local to the EJB server. Therefore it is much shorter lived, and there is no possibility of for example the client application crashing and leaving a dangling transaction in the server. Short lived transactions allow scaling to much higher workloads.

A sample client program illustrating the revised solution is shown in listing "B" at the end of this paper.

4.2 Parameter Object

Our *Part* entity bean is currently very simple. In a more realistic application it may have many additional fields. For example, category and sub-category, short and long description, stock level, supplier and supplier part number.

Imagine a client application that allows a user to enter a part number and see all the details relating to that part. The application would first find the appropriate *Part* bean using the *findByPrimaryKey* method on the home interface. It would then call the various accessor methods *getPrice*, *getDescription*, *getCategory*, *getSubCategory* and so on to populate the display. This results in a large number of remote method calls.

An alternative design is to introduce a new serializable Java class, say *PartData*,

which contains all the fields of a *Part* bean, and accessor methods to get them. Unlike the *Part* bean, the *PartData* object must be immutable – that is, it does not allow any of its values to be modified in any way. This is important to maintain data integrity, otherwise multiple clients could make inconsistent updates outside of the transactional control provided by the EJB server.

The *Part* bean is extended to include a *getPartData* method which returns a *PartData* object by value. A *setPartData* method may also be added which enables a client to make multiple updates with a single remote call as well. The *Part* bean will still retain the get and set methods for individual fields, since these are more efficient for a client wishing to query only a single value.

The *PartData* class that we introduced is sometimes known as a *Parameter* object or a *Value* object. Figure 4 illustrates this solution.

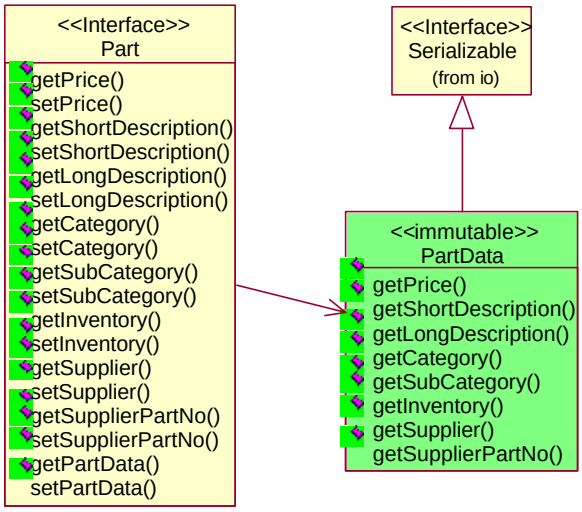


Fig. 4. Introduction of *PartData* as a parameter/value object.

Richard Monson-Haefel describes this approach as a "Bulk Accessor" in his "EJBNow!" website[14].

4.3 Enterprise Bean Granularity

Let us return to the relationship between an *Order* and its *OrderItems*. Currently both are implemented as entity beans. This means that every time an *Order* wants to manipulate its items, it has to do so through the *OrderItem* home and remote interfaces. This communication carries a higher overhead than direct inter-object communication, including the cost of all the container interposition discussed in section 1. As an entity bean there may also be performance issues related to locking and container managed persistence.

Representing *OrderItem* as an entity bean in its own right is a fine-grained approach to enterprise bean design. A coarser-grained approach turns the *OrderItem* from an entity bean into a *dependent object*.

A dependent object is one whose lifecycle is fully managed by another, independent object – in this case the Order entity bean. The EJB 1.1 [1] specification in section 9.1.2 defines a dependent object as follows:

"*A dependent object can be characterized as follows. An object B is a dependent object of an object A, if B is created by A, accessed only by A, and removed by A. This implies, for example, that if B exists when A is being removed, B is automatically removed as well. It also implies that other programs can access the object B only indirectly through object A. In other words, the object A fully manages the lifecycle of the object B.*"

and again,

"*A dependent object should not be implemented as an entity bean. Instead, a dependent object is better implemented as a Java class (or several classes) and included as part of the entity bean on which it depends*."

It is certainly true that an *OrderItem* cannot exist without the *Order* to which it belongs. With our design as it stands, the *Order* entity bean is responsible for creating (*addItem*), destroying (*removeItem*) and enumerating (*getItems*) *OrderItems*. However, *OrderItem* also includes an update method, *setQuantity*, that allows the state of an *OrderItem* to be updated independently of an *Order*.

Before we can turn *OrderItem* from an entity bean to a dependent object class we need to move this function to the *Order* entity bean. Why is this? If the update function were called independently of the Order entity bean, then no container interposition would occur, and the necessary transaction, persistence and security management would not be done. Dependent objects should be made immutable, and should be passed to the client by value.

Thus we remove the *setQuantity* method from the *OrderItem* interface, and turn *OrderItem* into a serializable class. We could either force a client to remove an item from an order before adding it again with a different quantity, or we could add an *updateItem* method to the Order bean that provides this behaviour. This solution is illustrated in figure 5.

Craig Larman discusses the topic in more depth in his description of the "Aggregate Entity Pattern" [15].

4.4 Collections and Iteration

Consider the Parts table which contains details about all of the parts, past and present, in our hypothetical company's catalogue. Over time this table could grow very large.

Suppose now we have a client application that needs to list all the parts sourced by a given supplier. Some suppliers may supply hundreds of different parts. How should we design our application to handle such a query?

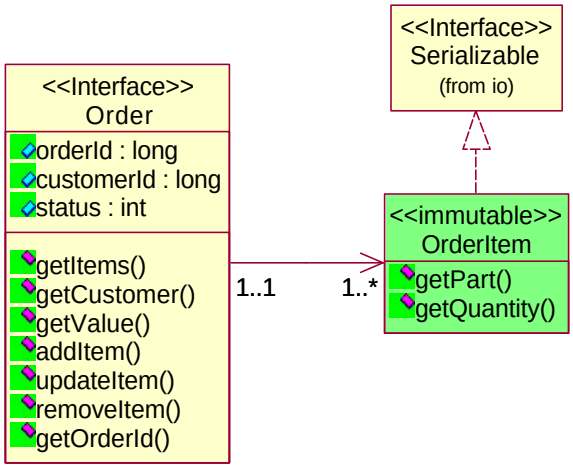


Fig. 5. OrderItem reworked as a dependent object.

The simplest design is to add a new finder method to the home interface of the *Part* entity bean:

```
public PartHome extends javax.ejb.EJBHome {

  java.util.Enumeration findBySupplier(
        long SupplierId )
  throws FinderException, RemoteException.
  …

}
```

The new finder method is required to return an enumeration of instances of the Part bean's primary key class.

A client application using this finder method would then have to iterate over the returned enumeration of primary keys. For each element it would invoke the Part home's *findByPrimaryKey* method, and then use the *Part* bean's remote interface to either get the *PartData* value object, or to query e.g. the short description only.

This technique causes a lot of remote calls between the client and the EJB server: after finding the home interface a further (1+2n) calls are required where *n* is the number of elements in the collection. It also creates *n* stubs on the client and *n* instantiated entity beans at the server – using up a lot of resources.

A more network efficient technique is to use a session bean to implement the query. A simple session interface might look as follows:

```
public interface PartFinder extends javax.ejb.EJBObject{

   PartSummary[] findPartsBySupplier(
     long supplierId ) throws RemoteException
  …
}
```

The *findPartsBySupplier* method defined in the *PartFinder* bean's remote interface returns an array of *PartSummary* objects. *PartSummary* is a value object that extends the Part primary key class to include the part's short description. The client can now iterate locally over the returned collection, displaying part number and short description for each element in the list. If a user selects one of the parts, the *PartSummary* object can be used to obtain the full details regarding that part in a call to *findByPrimaryKey* on the Part bean's home interface.

Figure 6 shows the various components in this solution.

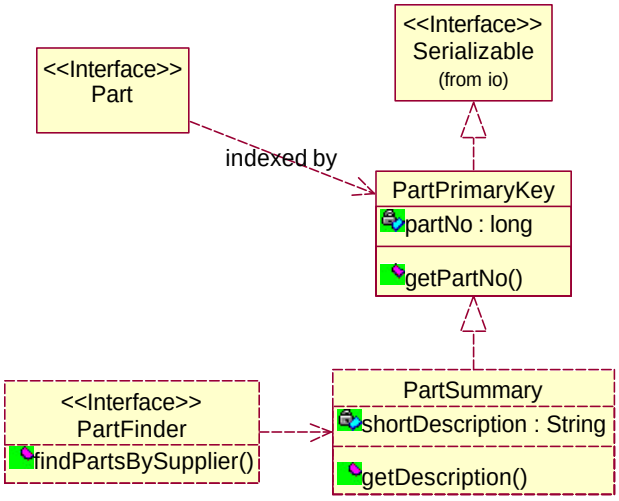


Fig. 6. The PartFinder solution.

5 OPEN CHALLENGES

The Enterprise JavaBeans specification is a great step forward for enterprise Java applications, and for component models in general. As ever though, there remain a number of open issues going forward that challenge both designers of EJB based systems and those responsible for taking the Enterprise JavaBeans specification itself forward.

5.1 Entity beans

The current version of the EJB specification does not provide any mechanism for detecting which enterprise beans have been modified within a transaction. This can result in unnecessary database updates. Some vendors have introduced proprietary extensions to overcome this problem, but a standard solution would be preferable. Advanced containers may introduce optimistic locking techniques to improve throughput.

In general, entity beans bring issues of object-relational mapping into the forefront of transaction processing systems. Converting data from object form to relational and vice-versa can be expensive. In a scalable, distributed system where a partitioned workload cannot be relied upon, the placement of object caches is difficult. Object database vendors may find a new lease of life applying their technology to object caching and persistence either in front of or instead of relational databases. An open container-persister interface would allow persistence vendors to plug their solutions into third-party containers.

More sophisticated container-managed persistence tools will be required to cope with coarser grained entity bean models where an entity bean does not simply map to a row in a single database table.

5.2 Integration with messaging

Work is underway to understand how best to integrate messaging with the Enterprise JavaBeans model. The benefits of integrating messaging and queuing in a transactional environment are well documented in Bernstein & Newcomer [16]:

- A client can submit a request even when the server is down
- A server can reply to a client when the client is down
- Communication failures do not result in lost replies or uncertain results
- Balancing a load amongst multiple servers is easier
- Priority can be given to some requests relative to others.

Bean-managed messaging allows beans to send and receive messages under application logic control. Container-managed messaging allows enterprise beans to be driven by incoming messages transparently to the enterprise bean application logic.

6 SUMMARY

The Enterprise JavaBeans specification defines a component architecture for distributed enterprise-level applications. Architecture and design of a distributed component based application raises different challenges and design points than for a non-distributed object-based application.

The façade or proxy pattern can reduce communication overhead between a client and a server, and can also reduce the time for which locks are held. It uses a session bean to represent the client locally within the server process.

The parameter or value object pattern permits passing of a set of related data in one remote method call instead of multiple calls, again reducing communications overhead.

Components are very powerful, but have an overhead associated with them due to container inter-position. Aggregating immutable dependent objects in a component can provide a more appropriate level of granularity and improve system throughput.

Iterating over a large collection of objects can tie up a lot of resources in the server and perform poorly due to the large amount of generated network traffic. Returning a set of value objects sufficient to display to an end user, and only fully instantiating

selected beans on which further processing is required can alleviate the problem.

The Enterprise JavaBeans specification continues to evolve. Challenges in object-relational mapping, locking and messaging integration are being addressed.

LISTING A. Client program to print shipping report before application of façade pattern

```
  …
  // Obtain a transaction context
  UserTransaction     tranCtx     =     (UserTransaction)initCtx.lookup(
"java:comp/userTransaction" );
  tranCtx.begin( );          // Start a new transaction


  // Obtain the order home
  Object objref           = initCtx.lookup( "OrderHome" );
  OrderHome                theOrderHome                                =
(OrderHome)javax.rmi.PortableRemoteObject.narrow(
           (org.omg.CORBA.Object)objref, OrderHome.class);


  // Locate the correct order
  Order theOrder = theOrderHome.findByPrimaryKey( new OrderKey( orderid
) );


  // Display the customer details:
  Customer theCustomer = theOrder.getCustomer( );
  System.out.println( "Customer Name:    " + theCustomer.getName( ) );
  System.out.println( "Customer Address: " +  theCustomer.getAddress( )
);


  // Display the ordered items:
  //  PartNumber    PartDescription             Quantity  PricePerUnit
TotalCost
  OrderItem[] orderContents = theOrder.getItems( );


  Part part;  int quantity; double partsCost=0.0,pricePerUnit=0.0;
  for (int i = 0 ; i < orderContents.length ; i++) {
    part         = orderContents[i].getPart( );
    quantity     = orderContents[i].getQuantity( );
    pricePerUnit = part.getPrice( );
    partsCost    = pricePerUnit * quantity;
    System.out.println( part.getPartNo( ) + " " + part.getDescription( )
```

```
+ " #" +
                            pricePerUnit + " x " + quantity + " = " +
partsCost );
  }
  System.out.println( "Total order cost = " + theOrder.getValue( ) );


  tranCtx.commit( );       // Commit transaction
```

LISTING B. Client program to print shipping report after application of façade pattern

```
  Object objref            = initCtx.lookup( "ReportHome" );
  ReportHome theReportHome = (ReportHome)
    javax.rmi.PortableRemoteObject.narrow((org.omg.CORBA.Object)objref,
ReportHome.class);
  Report theReport         = theReportHome.create( );
  String theReportData     = theReport.getShippingReport( orderId );
  System.out.println(theReportData);
```

ACKNOWLEDGEMENTS

The following are trademarks of International Business Machines Corporation in the United States, other countries or both: IBM, CICS, VisualAge and WebSphere.

Java, JavaServer Pages, JavaBeans, JNDI, JDBC and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries or both.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.

REFERENCES

1. Enterprise JavaBeans specification 1.1. Available at <http://java.sun.com/products/ejb/newspec.html>
2. IBM WebSphere Application Server. Details from <http://www.ibm.com/websphere>.
3. IBM CICS Transaction Server. Details from <http://www.ibm.com/software/ts/cics>
4. Java Servlets specification. Available at <http://java.sun.com/products/servlet>
5. JavaServer Pages (JSP) specification. Available at <http://www.javasoft.com/products/jsp>
6. Microsoft Transaction Server, Overview. Available at <http://www.microsoft.com/NTServer/appservice/exec/overview/Trans_Overview.asp>

7. Enterprise JavaBeans Technology vs. COM+/MTS. A collection of quotes and whitepapers available at <http://java.sun.com/products/ejb/ejbvscom.html>

8. Comparing Microsoft Transaction Server with Enterprise JavaBeans. Microsoft Corporation. <http://www.microsoft.com/ntserver/appservice/techdetails/techcompare/MtsJava.asp?

9. CORBA Component Model RFP. Available at <http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html>

10. IBM VisualAge for Java. Details from <http://www.ibm.com/software/ad/vajava>

11. Java Naming and Directory Interface (JNDI) specification. <http://java.sun.com/products/jndi/>

12. JavaBeans specification. Available at <http://java.sun.com/beans/docs/spec.html>

13. JDBC Data Access API. Available at <http://java.sun.com/products/jdbc>

14. EJBNow!, Richard Monson-Haefel. <http://www.ejbnow.com/ejbtips/html>

15. Enterprise JavaBeans 201: Aggregate Entity Pattern. (Technology Tutorial). Larman, Craig. Software Development, v8 n4 April, 2000.

16. Principles of Transaction Processing, Bernstein & Newcomer. Morgan Kauffman.