

New Life for EJB™*

By Rajat Taneja and Ganesh Prasad
(with grateful thanks to Vikrant Todankar and Simon Knudsen)

Version 1.2
3 August 2004

The proposed EJB 3.0 specification defines a new syntax to simplify development but fails to address fundamental flaws in the model.

Fortunately, there is a better way.

* EJB and all Java-related trademarks are the property of Sun Microsystems, Inc.

Revision History

<i>Version</i>	<i>Comments</i>
V 1.0	Initial draft
V 1.1	<ul style="list-style-type: none">- Changed Bulk Operation model so that developer code needs to be <i>either</i> an interface <i>or</i> an abstract class, rather than an interface <i>and</i> an abstract class implementing the interface.- Dropped the “getThis()” method from the Service Context. There is no need to identify a particular instance of a Service. They are all identical.- Added a table describing the development approach, in “A Better Model for EJB”- Added an extra example to Section 4.1.6- Added the point about empty Data Transfer Objects removing the need for multiple createXXX() methods in Section 6.3- Added e-mail addresses of authors.- “Outdented” some chapter headings to the right level.- Corrected miscellaneous typos.

<i>Version</i>	<i>Comments</i>
V 1.2	<ul style="list-style-type: none"> - Incorporated feedback from Evan Ireland of Sybase, and member of the EJB 3.0 spec committee - Incorporated feedback from Gavin King of Hibernate, and member of the EJB 3.0 spec committee - Modified tone of document to be more neutral - A major addition is the new model for data objects, which caters to Entities, DTOs and QueryTemplates - Many new sections to describe the new model - New, more intuitive names suggested for transaction types - Articulated argument against “magic” callbacks - Unmasked “stateful session bean” injected into client as a home interface in disguise - Articulated argument against detached entities - Articulated utility of the DTO model in exposing local/remote situations - Included Code Samples within main document - Added DTO and QueryTemplate Code Samples

Table of Contents

Revision History.....	2
EJB 3.0: Old <Wine> in New @Bottle.....	8
Problems with the EJB 2.x Model.....	10
A Better Model for EJB.....	11
Design Principles for a New Model.....	14
Refactoring the Model.....	16
Operations that belong at a higher level than a bean.....	16
Operations that belong within a bean.....	16
What's in a Name? Ambiguity and Uniqueness.....	18
Operations that belong at a higher level than a bean.....	18
Operations that belong within a bean.....	18
Detailed Description of the New Model.....	19
Identity.....	19
The Managed Component View.....	19
Dependency Injection or Service Locator?.....	19
The Lifecycle View.....	21
Container-side Lifecycle Events and Callbacks.....	21
Client-side Lifecycle Events and Managers.....	23
Clients Within the Container.....	23
Factories and Contexts.....	25
The Data View.....	26
Data Objects.....	26
The Entity Interface.....	26
Persistence – Container-Managed and Bean-Managed.....	27
Data Transfer Within and Across Tiers.....	27
DTOs that do not Map to Entities.....	31
Remote Creation of Objects.....	33
Queries and Bulk Operations.....	34
The Process View.....	35
Transactions – Container-Managed and Bean-Managed.....	35
State.....	36
The Event View.....	37
Messages.....	37
Timeouts.....	37
Things that Don't Belong.....	39
Security.....	39
Web Services.....	40
Design Artifacts for the New Model.....	41
Feedback on the EJB 3.0 EDR.....	42
Section 1.1 (Overview).....	42
Section 1.2 (Goals of this Release).....	42
Section 2.2 (Business Interfaces).....	44
Chapter 3 (Stateless Session Beans).....	44
Section 3.1.1 (Business Interface).....	44
Section 3.1.3 (Bean Class).....	45
Section 3.1.4 (Dependency Injection).....	47

Section 3.1.5 (Client View).....	47
Chapter 4 (Stateful Session Beans).....	48
Section 4.1.2 (Home Interface).....	48
Section 4.1.4 (Dependency Injection).....	48
Section 4.1.5 (Client View).....	49
Section 4.1.6 (Stateful Session Bean Removal).....	50
Chapter 5 (Message-Driven Beans).....	50
Section 5.1.1 (Business Interface).....	50
Section 5.1.3 (Dependency Injection).....	51
Section 6.1 (Requirements on the Entity Bean Class).....	51
Section 6.1.7 (Inheritance).....	51
Section 6.2 (Entity Manager).....	52
Section 6.2.1 (Entity Manager Interface).....	52
Section 6.2.2 (Example of Use of Entity Manager API).....	53
Section 6.3 (Entity Bean Life Cycle).....	53
Sections 6.3.1 to 6.3.4 (Creation, Removal, Synchronisation with the Database, Detached Entities).....	56
Section 6.3.5 (Transaction Rollback).....	56
Section 6.4 (Extended Persistence Context).....	57
Section 6.5 (Callback Methods).....	57
Section 6.7 (Security).....	59
Section 6.8 (Query API).....	60
Chapter 7 (Query Language).....	62
Chapter 8 (Entity Bean Context and Environment).....	62
Section 8.1.5 Universal Context.....	62
Chapter 9 (Metadata Annotations).....	63
Chapter 10 (Metadata for Object/Relational Mapping).....	63
Conclusion.....	64
Code examples.....	65
Sample 1 – Creating an Entity Bean.....	65
Step 1-1 – Declaring the Business Interface	65
Step 1-2 – Declaring the Entity Interface.....	66
Step 1-3 – Implementing the Entity Interface.....	67
Sample 2 – Creating a Service (Local client to the Entity).....	68
Step 2-1 – Declaring the Business Interface.....	68
Step 2-2 – Implementing the Business Interface.....	69
Step 2-3 – Telling the Entity Manager where to Look up the Entity.....	70
Sample 3 – Creating a Standalone J2SE Client (Remote client to the Service).....	71
Step 3-1 – Defining the client.....	71
Step 3-2 – Telling the Client-side Service Manager where to Look up the Service..	72
Sample 4 – Creating a Conversation (Local client to the Entity).....	73
Step 4-1 – Declaring the Business Interface.....	73
Step 4-2 – Implementing the Business Interface.....	74
Sample 5 – Creating a Web Tier Client (Remote Client to the Conversation).....	76
Case 5-1 – The web component holds the Conversation object in session scope.....	76
Case 5-2 – The web component maintains no state. The Conversation's handle gets sent all the way back to the browser.....	77
Sample 6 – Creating a true “Message-Driven Bean”.....	78
Step 6-1 – Declaring the Business Interface of a Service that should be invoked on	

receipt of a Message.....	78
Step 6-2 – Implementing the Business Interface.....	79
Step 6-3 – Defining the MessageListener class.....	80
Step 6-4 – Mapping the Listener to a Message Queue.....	80
Sample 7 – Creating a “Timer-Driven Bean” (A future-dated payment by a banking customer).....	81
Step 7-1 – Declaring the Business Interface of the Service that should be invoked on Occurrence of the Event.....	81
Step 7-2 – Implementing the Business Interface of the Service that should be invoked on Occurrence of the Event.....	82
Step 7-3 – Defining the TimeoutListener class.....	83
Step 7-4 – Declaring the Business Interface of the Service that sets the Timer.....	85
Step 7-5 – Implementing the Business Interface of the Service that sets the Timer..	86
Sample 8 – Creating Bulk Operations on a set of Entities.....	87
Step 8-1 – Defining a Custom Entity Manager Interface or Abstract Class.....	87
Step 8-2 – Implementing the corresponding EJB-QL in an XML file.....	88
Step 8-3 – Declaring the Business Interface for a Service Bean Client of the Bulk Operation.....	89
Step 8-4 – Implementing the Business Interface for a Service Bean Client of the Bulk Operation.....	90
Sample 9 – Implementing Queries on a Single Entity.....	91
Step 9-1 – Using a QueryTemplate Extracted from an Entity or its Business Interface Definition.....	91
Sample 10 – Distinguishing Original Objects from Copies.....	92
Step 10-1 – Declaring and Implementing the Business Interface of a Service that returns an Entity.....	92
Step 10-2 – Implementing a Local Client to the Service that Receives a Reference to the Entity itself.....	93
Step 10-3 – Implementing a Remote Client to the Service that Receives a Reference to a DTO (Copy of the Entity).....	94
Step 10-4 – Implementing an Application-Defined Data Transfer Object.....	95
Sample 11 – Implementing Persistence Interfaces.....	96
Case 11-1 – Implementing Basic Container-Managed Persistence.....	96
Case 11-2 – Implementing Container-Managed Persistence with Creation-time Callbacks.....	96
Case 11-3 – Implementing Container-Managed Persistence with all Entity Lifecycle Callbacks.....	97
Case 11-4 – Implementing Bean-Managed Persistence (with all Entity Lifecycle Callbacks).....	98
Sample 12 – Implementing Transaction Interfaces.....	99
Case 12-1 – Implementing Container-Managed Transactions in a Service Bean.....	99
Case 12-2 – Implementing Bean-Managed Transactions in a Service Bean.....	100
Case 12-3 – Implementing Container Transaction Lifecycle in a CMT Conversation Bean.....	101
Sample 13 – Implementing General Bean Lifecycle Interfaces.....	102
Case 13-1 – Implementing ResourceLifecycle (For Service Beans and Event Listeners).....	102
Case 13-2 – Implementing ConversationLifecycle (For Conversation Beans).....	103
About the Authors.....	104

About the Reviewers.....104
Appendix A – A Laundry List of Litanies (EJB 2.x)..... 105

Part I

EJB 3.0: Old <Wine> in New @Bottle

On June 30, 2004, the EJB 3.0 expert committee released an early public draft of the long-awaited specification. As long-time J2EE developers, we have had an opinion for some time now on what's wrong with EJB and the changes required in the model, so we eagerly grabbed the document to see if those issues had been addressed.

Overall, while there are some much-needed enhancements such as bulk update and bulk delete operations in EJB-QL, it looks like the committee has chosen to emphasise backwards-compatibility over clean design, and have therefore approached the complexity problem with a wallpaper strategy: *“Give developers a simpler programming interface, but don't mess with what's underneath.”* We find this approach disappointing, because the problems with EJB run far deeper, yet are very fixable.

We think this draft is more like an “EJB 2.2” specification, because it defines little more than a wrapper around the old model. Whether one employs XDoclet's mongrel approach or annotations with a JSR 175 pedigree, it's still nothing more than a workaround, because the core architectural issues of that model remain unaddressed. A true “3.0” spec should be bold enough to confront these architectural issues and fix them, even if it breaks backwards-compatibility with older applications.

At the same time, many of the vaunted innovations in the spec, such as the use of annotations and dependency injection are, we believe, overdone. We think EJB can be made to work cleanly and elegantly without an overreliance on these mechanisms. We also find that the persistence model in the spec has not been well thought out. The most egregious aspect of this model is the new “detached” mode for Entity Beans, which is based on invalid assumptions and is a clumsy solution to a simple problem. It will in fact create more problems than it solves. We realise these are strong assertions to make, and we provide justifications for all of them in this document. We have also provided this feedback to the committee itself.

Our stand is that complexity must be *removed* from EJB, not just hidden from view by annotations and other metatags. In fact, annotations don't always hide complexity, as anyone who has seen the proposed persistence mapping annotations can attest. We certainly don't want additional complexity in the form of new concepts and new syntax elements. Come back, deployment descriptors, all is forgiven! ;-)

We must hasten to clarify that unlike many others in the Java developer community, we are not EJB haters.

We believe in the managed component model, where the container takes responsibility for the bulk of an application's infrastructural requirements, leaving the application developer free to concentrate on pure business logic. We believe that CMP Entity Beans represent the most advanced persistence model available, although the clumsiness of their implementation as mandated by earlier specifications has made the use of Entity Beans needlessly complex and hence deeply unpopular. (Our model of Entity Beans removes this complexity.)

We view Hibernate and JDO as possible mechanisms for containers to *implement* CMP, but

we don't seriously expect them to *replace* Entity Beans. These frameworks know *how* to persist an object, but only the container knows *when* to persist it (based on transaction boundaries), which is why most Entity Bean-bashing is naive and misinformed.

We also believe there is a legitimate and important role for Stateful Session Beans in enterprise applications. They provide a more robust and channel-independent session management mechanism than the HttpSession-based equivalent familiar to web-tier developers.

We like the choice of container-managed and bean-managed mechanisms for both persistence and transactions, but prefer the container-managed option wherever possible.

We believe that deployment descriptors are the best way to express certain kinds of metadata, but that deployment descriptor diarrhoea must be urgently treated.

In short, our disagreements are not on the *scope* of the EJB model, but on its design. The EJB 2.x model is a work of art – a Picasso, in fact. All the required parts are there – in all the wrong places!

Problems with the EJB 2.x Model

Like most J2EE developers, we have had our share of problems getting our heads around EJB concepts. After months of demoralising self-doubt relating to IQ, we each independently came to the conclusion that the problem was not with us at all. It was the EJB model that was unnecessarily convoluted and counter-intuitive.

Here is a quick listing of the fundamental things we think are wrong with the EJB model as it stands today (EJB 2.1):

- Flaw 1. It forces Entity Beans, Stateless Session Beans and Stateful Session Beans to implement common interfaces when they are in fact very different animals. These interfaces are simultaneously anaemic and overloaded, forcing implementations to either repeatedly provide the same functionality, or implement irrelevant functionality.
- Flaw 2. It fails to separate operations *on a set of beans* from operations *on a single bean*, forcing an individual bean to implement functionality relevant to itself as well as functionality affecting a set of similar beans.
- Flaw 3. It creates unnecessary dependencies between components, breaking encapsulation in a number of places.
- Flaw 4. It overloads terms with multiple meanings in different contexts, confusing developers.

These flaws are not fatal (because EJBs can be made to work quite satisfactorily), but they are responsible for the “complexity crisis” that EJB finds itself in today. That's the crisis that the EJB 3.0 spec is attempting to solve in its superficial, let's-not-rock-the-boat kind of way.

We are convinced that all the problems developers face with EJBs can be traced back to one or more of these fundamental design flaws, and therefore that the solution to these problems lies in refactoring the model to address these flaws. To avoid cluttering the article, we elaborate on the problems with EJB 2.x in Appendix A, and move directly on to our proposed solution.

Part II

A Better Model for EJB

“Make things as simple as possible, but no simpler” - Albert Einstein

In a sentence, the model we present is nothing but a refactored EJB 2.1 model, minus aspects that don't belong (e.g., Security and Web Services).

We find that our refactoring immediately solves the four fundamental issues identified in the previous section, and makes EJB development simple and intuitive, as the attached code samples will show.

A comparison of our approach against that of the EJB 3.0 spec highlights the following:

1. As opposed to the number of resources injected into beans in the proposed EJB 3.0 model, there is only one instance of dependency injection in ours (which is exactly the same as in EJB 2.x). This is where the container sets the bean's context through an appropriate `setXXXContext()` method. We contribute a tiny bit of cleverness here by providing access to Service Locators from within this injected context itself. A bean can now look up all the resources and components it requires by calling simple getter methods on the Service Locators that it already has access to through its context. Lookups are therefore drastically simplified. JNDI lookups are not required to be seen by the developer, although the framework may use it under the covers.
2. For non-EJB clients, we describe a vendor-neutral Service Locator factory with the smarts to download app server-specific implementations of remote Service Locators. These can be used, in turn, to look up remote resources through simple getter methods as before. Clients of the EJB tier do not need to know about JNDI either, nor are they dependent on specific libraries to be provided by an app server vendor. In contrast, the EJB 3.0 spec does not go into detail on how non-EJB clients of the EJB tier will find and call EJBs.
3. Our component model is simple and intuitive, so there is no need to use annotations to hide complexity and very limited need to provide extra information to the container. Regular Java syntax is sufficient to express most aspects of an application, reducing even the need for deployment descriptors. The need for annotations is therefore drastically reduced in our model. Annotations can still be used, of course, but they add value in far fewer situations because the model itself is now much cleaner.
4. We also confront one of the fundamental problems with distributed systems that try to hide the distributed nature of their components – the thorny issue of pass-by-reference versus pass-by-value that bites developers precisely because the local-remote dichotomy is too well hidden. We solve the problem not by forcing the EJB model (and developers of client applications) to declare and use explicit local or remote interfaces for beans, but by defining a new marker interface that the framework automatically applies to copies of objects that it creates for remote clients. Remote clients can now easily tell if they are dealing with a copy of an object or the original, reducing the possibility of errors. This is the enabling foundation for Data Transfer Objects in our model.

In contrast, the EJB 3.0 spec starts with the highly questionable objective of exposing the domain model itself to other tiers (a strategy of tight coupling!) and compounds the problem by apparently confusing Data Transfer Objects with Entities, based on superficial similarities. Pursuing this logic, it goes on to propose “detached entities” for use as DTOs and builds the concept into the entity bean lifecycle itself as “detach” and “merge” events, instead of just taking snapshots of entities and using them as DTOs! Finally, it couples the client tier once more with the EJB tier by requiring the client to “evict”, “flush” and “refresh” entities.

We reject this entire model as clumsy, unnecessary and problematic. We treat Entities as Entities, and DTOs as DTOs. The much-maligned business interface comes to our rescue as a shared interface for an Entity and a DTO, which otherwise have nothing in common. Thus, the properties corresponding to an entity can be conveniently transported, but the transport envelope itself (the DTO) is never mistaken for the Entity. We have separate mechanisms for each (including well-defined ways to create a DTO that maps to an Entity's properties, and to update an Entity with properties in a corresponding DTO). Our entities are always managed (never detached), making the container-side lifecycle of entities much simpler and decoupled from events in other tiers.

5. Our programming model emphasises interfaces. EJBs play a variety of roles simultaneously, and need to expose different faces to different audiences, so to speak. The EJB 2.1 model's grouping of methods into interfaces is incorrect and makes EJB development unnecessarily complex. Our model corrects this with a better grouping that makes the structure of EJB far more understandable. Additionally, such grouping exposes the optional nature of some of these interfaces, and they can be ignored by beans that don't require the functionality.

In contrast, the EJB 3.0 spec seems to view the number of interfaces as something to minimise, and tries to drive development, as far as possible, through implementation classes alone.

This approach is especially obvious with Entity Beans, where the spec bravely asserts that no business interface is necessary, and then has to resort to “detached entities” to implement Data Transfer Objects. Our insistence on business interfaces even for Entity Beans allows us to cleanly implement DTOs that act as snapshots of Entity state.

The spec also pretends that Stateful Session Beans require no home interface, but injects one into a client anyway, in the guise of the bean itself.

The spec's approach of minimising interfaces also shows up in its “magic” callback mechanism, under which beans can implement lifecycle callback methods *without* implementing their containing interface, and have the container automatically recognise and call them. We think this can lead to some interesting situations in the future, when newer callback method names match those of existing regular methods and result in some rather unexpected behaviour. In our model, we follow the standard Java interface discipline. Our classification keeps the number of interfaces and callback methods reasonable and pertinent to the functionality that a bean is trying to implement, so the requirement to fully implement all methods of a declared interface isn't too onerous.

The following table shows the approach to be used for each type of component:

<i>Type of component</i>	<i>Developer's Responsibility</i>
Entity Beans, Stateless Session Beans and Stateful Session Beans	Write a Business Interface and a Bean class (implementation) that implements it and other interfaces.
Event Listeners (Message and Timer)	Write only the Event Listener class (implementation) that implements a well-known Event Interface.
Custom Entity Managers	Write only an Interface or Abstract Class. The container will generate the implementation.

6. Our model uses extremely simple Java syntax and the design involves no concept more complicated than interfaces, implementations and callbacks. It can be readily understood by Java developers who have learned Java 1.2 (which includes the Collections API). The EJB 3.0 model, with the stated aim of reducing complexity, is nevertheless introducing a learning curve of its own by requiring developers to learn basic annotation grammar and a large annotation vocabulary, along with concepts such as dependency injection, detached entities, a more complex entity bean lifecycle, the new notion of “persistence context”, and others before they can start developing applications.

As can be seen, our model does not require “sexy” new technology buzzwords like AOP or lightweight containers, nor do we need large-scale use of annotations or dependency injection to make development simpler. The model is inherently simple, yet sacrifices none of the power of EJB.

This is not “EJB Lite”. We like to think of it as “EJB done right”.

Design Principles for a New Model

Our model is based on these principles:

1. Entity Beans, Stateful Session Beans and Stateless Session Beans are first-class components in their own right and are recognised as such.
2. A component (of any of the above types) only defines operations that are relevant for itself (its instance) and not for a group of components.
3. Operations on *groups* of components are the responsibility of specialised Manager classes. There is one Manager class for *each* type of component (Entity, Stateful Session and Stateless Session).
4. Some of these operations (on groups of components) are standard operations and are the same for all components of the same type (Entity, Stateful Session or Stateless Session). These tend to be component lifecycle operations and are provided by the EJB framework itself. It implies that lookup logic can be subsumed by the framework and become invisible to the developer.

Other operations (and these happen to apply only to Entity Beans) are peculiar to the particular subclass of component, i.e., they are custom operations. These are typically bulk operations and will have to be written by the application developer, – but inside a manager, not inside a bean! If the method needs to be expressed in EJB-QL, it is written inside an XML file and the manager only exposes the interface of the method. If it is in Java, it is written inside the manager itself. Accordingly, the manager is either an interface or an abstract class, with the concrete implementation provided by the container.

5. An individual component itself has several kinds of contracts that it needs to honour. Though this may seem complex at first, the classification is simple and intuitive.
 - To start with, there is an *identity* interface that a component must implement to advertise its type to both clients and the container. The identity interface may be just a marker, or it may define behaviour that a component of this type must exhibit to both clients and the container.
 - Since this is a managed component, there is an additional, mandatory *container-facing* interface that it must implement, based on its type. The container uses this interface, but clients should not have to know about it. That's why this is distinct from the identity interface.
 - Then there is the specific *business* interface that distinguishes this component from all others of its type and forms its contract with the client. This is actually implemented by the component, unlike in EJB 2.x, where it is merely “shadowed”.
 - Finally, there are a number of *optional* container-facing interfaces that the component may implement, depending on its type. These optional interfaces could be additional indicators to the container, or mechanisms for the container to provide callbacks to the component at certain points in its lifecycle.
6. There is a new Event Listener component, distinct from Entity Beans, Stateful Session Beans and Stateless Session Beans. The purpose of such components is to listen to asynchronous events such as messages and timeouts, to unmarshall data embedded within

such events, and to trigger business logic in Stateless Session Beans, passing in this data as method parameters. Message-Driven Beans can be readily decomposed to fit into this model, but the EJB 2.1 Timer mechanism needs to be completely rearchitected to do so.

7. All class, interface and method names are unique and intuitive.
8. Entities are only visible to clients within the container, along with Event Listeners and Timers. Stateless Session Beans and Stateful Session Beans may be accessed within the container (local) or by clients outside the container (remote). The actual nature of the communication (local or remote) is not visible to the client. Data Transfer Objects are explicitly identified as such, so there should be no danger of clients assuming pass-by-reference semantics when pass-by-value is in use, and vice-versa.
9. Entity Beans, Stateless Session Beans and Stateful Session Beans are all proxied, all the time. Their clients never receive a reference to the beans themselves, only to their proxies. This is true whether the access is local or remote. The client-side stub transparently uses an appropriate mechanism to communicate with the bean.

Refactoring the Model

Some quick classification tells us where different methods really belong.

Operations that belong at a higher level than a bean

Interface	<i>Entity Bean</i>	<i>Stateless Session Bean</i>	<i>Stateful Session Bean</i>	<i>Message-Driven Bean</i>
Client-side lifecycle interfaces	create findByPrimaryKey remove	create	create remove	-
Client-side bulk operations	findXXX home-BusinessMethod	-	-	-

Operations that belong within a bean

Interface	<i>Entity Bean</i>	<i>Stateless Session Bean</i>	<i>Stateful Session Bean</i>	<i>Message-Driven Bean</i>
Identity Interfaces	getPrimaryKey	-	getHandle	-
Business interfaces	accessors	businessMethods	businessMethods	-
Event interfaces	-	-	-	onMessage
Container-facing mandatory interfaces	setEntityContext	setSessionContext	setSessionContext	setMessage-DrivenContext
Lifecycle callback interfaces	ejbCreate ejbPostCreate ejbRemove ejbLoad ejbStore ejbActivate ejbPassivate	ejbCreate ejbRemove	ejbCreate ejbRemove ejbActivate ejbPassivate <u>From Session-Synchronization:</u> afterBegin beforeCompletion afterCompletion	ejbCreate ejbRemove

The EJB model already looks more logical and understandable when classified in this way. If these methods are placed in separate interfaces and implemented only by the components to which they apply, 90% of our refactoring is done! The next step is to improve the interface and method names to be more meaningful.

E.g., we don't like the terms "Stateful Session Bean" and "Stateless Session Bean". One is a tautology, the other an oxymoron. Sessions are always stateful! We call our stateless and stateful components Service Beans and Conversation Beans, respectively. Perhaps "Service"

and “Session” would be more appropriate, but the term “Session” is now polluted and likely to confuse.

We are also demoting “Message-Driven Bean” to the status of a composite component, not a first-class EJB. The business logic goes into a Service Bean, and the remainder becomes an Event Listener. Event Listener subclasses can now listen to messages as well as timer events, and call Service Beans to trigger business logic, so the model for messages and timer events is now the same. Event Listeners are also managed components that are stateless like Service Beans, and so the container similarly pre-allocates them into pools.

The method names need a fair bit of work too, especially the lifecycle methods on the client and container sides. E.g., the method “create()” on the client side means different things for different beans. For Entity Beans, it means a request to create a new permanent record. For Stateless Session Beans, it's just a request to acquire any Service component at all from the pool of pre-allocated components. And for Stateful Session Beans, it means a request to initiate a new session that maintains state across more than one request. The methods should more properly be named “createEntity()”, “getService()” and “initiateConversation()”.

As an example of server-side lifecycle events, we again don't like the term “ejbCreate()” being used for three different types of beans when they mean different things altogether. For Entity Beans, “ejbCreate()” is called before a row is inserted into a table (“ejbPostCreate()” is called soon after). For Stateless Session Beans, “ejbCreate()” is called after the bean is initially created by the container and placed in a pool. It has nothing to do with a client's request for a bean, and occurs even before any clients connect to the application server. For Stateful Session Beans, “ejbCreate()” is called after the bean is created in response to a client's “create()” request. This overloading of method names is inexcusable. We prefer the terms “beforeCreate() and afterCreate()” for Entity Beans, “afterAllocate()” for Services (Stateless Session Beans) and “afterInitiate()” for Conversations (Stateful Session Beans).

What's in a Name? Ambiguity and Uniqueness

The tables below show the newly-reclassified interfaces and methods with new names.

Operations that belong at a higher level than a bean

Interface	<i>Entity Bean</i>	<i>Service Bean</i>	<i>Conversation Bean</i>	<i>Event Listener</i>
Client-side lifecycle interfaces	createEntity findEntity removeEntity	getService	initiateConversation findConversation endConversation	-
Client-side bulk operations	findXXX home-BusinessMethod	-	-	-

Operations that belong within a bean

Interface	<i>Entity Bean</i>	<i>Service Bean</i>	<i>Conversation Bean</i>	<i>Event Listener</i>
Identity Interfaces	getPrimaryKey	-	getHandle	-
Business interfaces	accessors	businessMethods	businessMethods	-
Event interfaces	-	-	-	onMessage, onTimeout (in subclasses)
Container-facing mandatory interfaces	setEntityContext	setServiceContext	setConversation-Context	setEventListener-Context
Lifecycle callback interfaces	beforeCreate afterCreate beforeDestroy afterLoad beforeSave inactivity-PeriodEnded inactivity-TimeoutOccurred	afterAllocate beforeWithdraw	afterInitiate beforeEnd afterResume beforeSuspend (From Conversation-Transaction-Lifecycle:) afterTransaction-Begin beforeTransaction-End afterTransactionEnd	afterAllocate beforeWithdraw

Part III

Detailed Description of the New Model

Identity

Every EJB (as well as a couple of other component types) must implement an interface that advertises what it is. Sometimes, this identity interface also declares methods that a component of this type must implement.

Here are the identity interfaces in our model:

- **Conversation** (Stateful Service Bean): This declares the `getHandle()` method, through which a client can obtain a serialisable handle that can be used to resume a paused transaction later (even from another client device).
- **Service** (Stateless Session Bean): This is a marker interface.
- **Event Listener** (The “event listener” part of Message-Driven Bean and a logical superinterface of `TimedObject`): This is a marker interface.
- **Entity**: This declares the `getPrimaryKey()` method.
- **DTO**: This is a marker interface that framework-created objects implement to show that they are copies of other objects, either replicated remotely or replicated from Entities.
- **QueryTemplate**: This has the `addCondition()` method that allows configuration of queries.

The Managed Component View

The four main managed components in our model (Entities, Services, Conversations and Event Listeners) require different services from their environment to do their work. The exact services and resources they need vary, but the general pattern by which they relate to the container is the same. We have a choice of mechanisms. The container can inject all required resources into a bean, or the bean can look them up.

Dependency Injection or Service Locator?

In this context, there is an excellent paper by Martin Fowler called “Inversion of Control Containers and the Dependency Injection Pattern”, which is available at <http://www.martinfowler.com/articles/injection.html#ServiceLocatorVsDependencyInjection>.

The basic point Fowler makes is that both patterns (Service Locator and Dependency Injection) provide useful decoupling between components. Service Locator is the more straightforward mechanism of the two, and more easily understood. However, it implies a new dependency upon the Service Locator itself. Designers need to ask themselves if that is a problem. Dependency Injection tends to be harder to understand, and designers need to ask themselves if the extra complexity is justified. So there's a trade-off.

In our model, a bean uses Service Locators exclusively to locate other beans and enterprise resources, because this is simple and straightforward. At the same time, we eliminate the bean's dependency on the Service Locators by having them injected through the bean's Context. In fact, the Context is the only thing injected into a bean. This makes sense to us, because the Context is meant to be the bean's only connection back to the container. By injecting the Context, we remove the explicit dependency of the bean upon the container.

Examples:

```
...
ResourceManager rm = context.getManagerFactory().getResourceManager();
DataSource ds = rm.getDataSource( "PostgreSQL Testing DB" );
....
EntityManager em = context.getManagerFactory().getEntityManager();
MyEntity myEntity = (MyEntity) em.findEntity( MyEntity.class, primaryKey );
...
ServiceManager sm = context.getManagerFactory().getServiceManager();
SomeService someService = (SomeService) sm.getService( SomeService.class );
```

We believe this approach is far cleaner than having a dozen different objects injected into a bean. In our model, the lookup is extremely simple, involving ordinary Java getter methods. Any JNDI that is employed by a Service Locator is completely hidden from the bean's (and therefore the bean developer's) view.

[Incidentally, Fowler characterises the EJB framework as “very intrusive”. We wonder if our model fixes that.]

Entities only require access to other Entities, so their context only gives them access to Entity Managers. Event Listeners, Services and Conversations can potentially access any resource, so their context gives them access to a ManagerFactory, using which they can acquire references to any other Manager component. Event Listeners, Services and Conversations also get access to a TransactionController, using which they can control aspects of their transactions. Entities cannot control transactions. Event Listeners, CMT (Container-Managed Transactions) Services and CMT Conversations only get access to a basic TransactionController object that lets them mark a transaction for rollback and to check if another component has already marked the transaction for rollback. BMT (Bean-Managed Transactions) Services and BMT Conversations get access to a more fine-grained transaction controller that lets them specify when transactions begin, when they should be committed, when they should be rolled back (immediately, not just marked for rollback), what the transaction timeout should be, etc., and find out the detailed status of a transaction.

The type of TransactionController (basic or fine-grained) is automatically placed in the bean's context depending on whether the bean has implemented the marker interface ContainerManagedTransaction or BeanManagedTransaction. Event Listeners only get access to the basic TransactionController.

The Lifecycle View

EJBs are managed components, which means that the container does many things for them without their having to lift a figurative finger. Different types of beans have different lifecycles, because different things are done for them by the container. However, there are opportunities for a bean to intervene and do some processing of its own just before (or just after) the container does something to it. These are called lifecycle callbacks.

Container-side Lifecycle Events and Callbacks

Entity Lifecycle

Entities are created and removed. They are loaded up from persistent storage, and saved back to persistent storage. After a period of inactivity, the container may put an entity back in a pool to conserve resources, and pull it back when some activity occurs. These are the major Entity Bean container-side lifecycle events.

We classify them as follows:

EntityCreationLifecycle:

beforeCreate()
afterCreate()

EntityLifecycle:

Creation lifecycle, plus
beforeSave()
afterLoad()
beforeDestroy()
inactivityTimeoutOccurred()
inactivityPeriodEnded()

We split them this way because some Entity beans may want to receive creation-related callbacks alone and don't want to be bothered by any others.

Both EntityCreationLifecycle and EntityLifecycle are optional for CMP Entity Beans. BMP Entity Beans must implement a marker interface (BeanManagedPersistence) that extends EntityLifecycle, so they must implement all these callback methods.

Service and Event Listener Lifecycle

Services and Event Listeners are stateless components that the container pre-allocates into pools on system startup. One bean is identical to another of its type. Their lifecycle consists of just being allocated and pooled, and being removed from the pool to conserve resources. Accordingly, their callback methods are:

ResourceLifecycle:

afterAllocate()
beforeWithdraw()

The ResourceLifecycle interface is optional.

Conversation Lifecycle

Conversations are stateful components that need to hold on to state for the duration of a client conversation. They are initiated, but may be suspended to conserve resources if the conversation is idle for too long. Conversations can be resumed, and finally ended, when the state held by the bean can be thrown away.

Their callbacks are:

ConversationLifecycle:

afterInitiate()
beforeSuspend()
afterResume()
beforeEnd()

Conversations that have their transaction managed by the container can request callbacks on transaction-related events, specifically, just after a transaction begins, before it ends and after it ends (successfully or unsuccessfully). This is nothing but the SessionSynchronization interface with better (we think) names.

The callbacks are:

ConversationTransactionLifecycle:

afterTransactionBegin()
beforeTransactionEnd()
afterTransactionEnd() (returns a boolean to say whether the transaction succeeded or not)

The ConversationTransactionLifecycle and ConversationLifecycle are both optional and can be implemented independently of each other.

Client-side Lifecycle Events and Managers

Clients do different things with EJBs, quite apart from using them by calling their methods. Creating, finding and removing EJBs are operations that come under the category of client-side lifecycle operations. Unlike in the EJB 2.x model, where such operations were linked to an instance of a bean itself, we explicitly propose a new class of component called Managers, one for each kind of EJB.

Service Lifecycle

The Service lifecycle is the simplest of all. The client requests a reference to a service and makes a call on it. If the client holds on to the reference and makes another call, it doesn't necessarily mean that the same container-side Service Bean is going to handle the request. It could be another one from the Service pool. They're all identical.

There's no point in the client telling the manager that they're done with the Service, because the manager (and the container) will both shrug, and the Service will continue in the pool. Services get withdrawn from pools for reasons that have nothing to do with the client.

ServiceManager:

getService(): takes a business interface and returns a reference to a Service

Conversation Lifecycle

Conversations are more complex than Services. They need to be set up, and a Conversation Bean is created directly in response to a client request. Because Conversations eat up resources in proportion to the number of clients that have initiated them (regardless of how many are active), the container will have to do smart things to conserve resources (as we saw in the container-side lifecycle). From the client's side, though, a cool feature that Conversations offer is the ability to resume a conversation from another client device altogether. Conversations provide a "handle", which can be saved by clients and used to find them again. It's good practice for clients to inform the container (through the manager) that they're done with a Conversation Bean, because the container can then reclaim the resources used by it.

ConversationManager:

initiateConversation(): takes a business interface and returns a new Conversation

findConversation(): takes a business interface and a handle and finds the matching
Conversation

endConversation(): tells the container to remove the given Conversation

Clients Within the Container

Services and Conversations are the only EJBs a non-EJB client can see, and consequently, Service Managers and Conversation Managers are the only manager components a non-EJB client should have to deal with.

But components within the container may need to do more. They need to deal with Entities, Timers and other resources.

Entity Lifecycle

An Entity Bean's client is either a Service or a Conversation. Sometimes, another Entity Bean may require to call lifecycle management services on it. Event Listeners should not access Entities directly.

These are the methods that an EntityManager provides:

EntityManager:

createEntity(): can take either the business interface or a DTO to create the entity

findEntity(): takes the business interface and the primary key to find the entity

findEntities(): takes a query template to find a set of matching entities

updateEntity(): takes a DTO and the primary key and updates the entity

removeEntity(): removes the given entity

We don't have a remove() method on the bean that a client can call to remove it. We think it's cleaner to do all lifecycle management through the manager.

The manager is also a good place to put other operations that are “above” the level of an individual bean, in the sense that the bean developer shouldn't have to write it afresh for each bean.

getDTO(): can take either the business interface or an entity and return a DTO

getQueryTemplate(): can take either business interface or entity and return a query template

areIdentical(): takes two entities and says if they're the same

Timers

Server-side components (especially Services and Conversations) may need to set timers that will result in some processing being triggered at a future point in time. Event Listeners should not have to set timers themselves, but they could. Entities should not set timers. Timers need to be managed through a manager component too.

TimerManager:

createTimer(): returns a Timer

findTimer(): takes a handle and finds an existing Timer

removeTimer(): removes a given Timer

getTimers(): returns a Collection of all Timers being managed by the system

Resource Adapters and DataSources

Server-side components (Services, Conversations and Event Listeners) may need access to resources through a Resource manager, although Event Listeners should ideally delegate such processing to Services.

ResourceManager:

getDataSource(): takes a name and returns a DataSource

getResourceAdapter(): takes a name and returns a resource adapter

Factories and Contexts

All the managers described above are convenient to use, but they need to be acquired as well. What is required is a manager factory to provide access to all of them in one place.

ManagerFactory:

getTimerManager()

getServiceManager()

getConversationManager()

getResourceManager()

getEntityManager(): returns a generic entity manager or a custom subclass if a business interface is given

An implementation of the manager factory is put into the context of Services, Conversations and Event Listeners, so they can acquire all required resources and components from one place.

Non-EJB clients in other tiers should not have to see any components other than Services and Conversations, and therefore they should see no manager components except ServiceManager and ConversationManager. But non-EJB clients have no context that can hold a reference to a manager factory. So the client-side ManagerFactory has to be provided as a concrete class with a factory method.

client.ManagerFactory:

getInstance(): static method, returns a reference to an instance of the factory

getServiceManager()

getConversationManager()

The Data View

Data Objects

Entities are not the highest level of abstraction possible for data. A data object is a concept by itself, and it is defined by nothing more than a set of properties (with getters and setters). This is what we call the Business Interface. Data objects can manifest themselves as many things in different contexts:

- An Entity is a data object that is persistent and managed. It has to have a primary key to distinguish it from the multitude of other managed entities with the same business interface.
- A Data Transfer Object (DTO) is a data object that is meant to wrap a set of attributes in a convenient envelope, suitable for operations such as transmission across a communication medium. It has no special methods and no requirement for a primary key, but it must be Serializable.
- A Query Template is a data object that is meant to find objects that match the values of its properties (Query By Example). It may have additional methods to specify conditions on its properties.

These three interfaces (Entity, DTO and QueryTemplate) are what we will call Identity Interfaces (which tell us what an object **is**, as opposed to the Business Interface, which tells us what it **has** or **does**).

The Entity Interface

Both clients and the container need to see the business interface as well as the identity interface for a data object playing the role of an Entity (Containers see the actual implementation as well). Since it would be a pain for the client or container to have to cast the object to one or the other interface to access its properties on the one hand and its primary key on the other, we would need to either make the business interface extend the identity interface, or define a new subinterface that extends both, and expose this subinterface to both clients and the container. Either way, both the properties and the primary key should be accessible from the same interface.

The second approach is more verbose than the first but also cleaner and more flexible, because it allows developers as well as the container to create objects that implement the business interface and one of the other identity interfaces, without having to bother about a `getPrimaryKey()` method that doesn't belong. This is the approach we recommend and use in our sample code. This subinterface is called the Entity Interface.

[We also suggest a syntactic shortcut using static inner interfaces so the developer isn't forced to create two interface files per Entity. See the code examples.]

Persistence – Container-Managed and Bean-Managed

Entities represent persistent data objects. They are also managed. Entity Beans must implement one of two persistence-related interfaces that tell the container how to manage the Entity.

If the Entity implements `ContainerManagedPersistence`, then the container will perform all entity lifecycle operations on the bean without giving it any callbacks (i.e., any opportunities to perform processing just before or after such an operation). A CMP Entity Bean is of course free to implement `EntityLifecycle` to receive all such callbacks, or just implement `EntityCreationLifecycle` to get the two creation callbacks alone (`beforeCreate()` and `afterCreate()`). This may be useful even for CMP Beans, because they may need to generate a primary key and any non-null fields before the creation, and to set relationship mappings after creation.

If the Entity implements `BeanManagedPersistence`, then it must implement all the callback methods in `EntityLifecycle`. It can choose to do nothing within those methods, but all events will be notified to the bean.

Data Transfer Within and Across Tiers

DTOs are objects that are transported between tiers. But the DTO interface in our model is not meant for developer-written classes to implement, even if the developer is writing a data transfer object! The purpose of the DTO interface is to identify objects that have been transparently serialised and copied across to another tier by the *EJB framework*. The framework recreates the object on the other side, and makes it implement the DTO interface. It's an admission of guilt by the framework ("This isn't the real thing I'm giving you"). If the object was passed by reference, then it doesn't implement the DTO interface when it is received.

If the data object that is passed between components is an Entity, and the components are in the same tier, then the receiving component gets a reference to the Entity itself. If the receiving component is in another tier, the Entity is serialised and recreated as an object implementing the same business interface and the DTO interface. It is now clearly not an Entity anymore, although it has the same accessors. For this mechanism to work, Services and Conversations must not declare the return type of any method as a subclass of Entity, but as the business interface. Also, the business interfaces themselves should avoid extending the Entity interface (otherwise, the DTO will feature a rather curious `getPrimaryKey()` method that returns a null).

DTOs are thus created implicitly by Services and Conversations. If a Service has a method returning entity references, and declares that method as returning a Business Data interface (or a Collection of Business Data interfaces), then what a local client should actually receive is an Entity (or a Collection of Entities), and what a remote client should receive is a DTO (or a Collection of DTOs). This is the usual pass-by-reference versus pass-by-value that the EJB 2.0 spec distinguishes through local and remote interfaces.

This example shows how it works:

```
public interface ProductService extends Service
{
    // Declares the return type as the Business Interface, not the Entity Interface
    public Product getProduct( String _productId );
}

public class ProductServiceBean
implements    ProductService,           // for the business method getProduct()
             ServiceBean,              // for setServiceContext()
             ContainerManagedTransaction // to receive a minimal
                                         //transaction controller
{
    @Nontransactional // Or use Deployment Descriptor
    public Product getProduct( String _productId )
    {
        ...
        // What the finder returns is an Entity interface.
        // The implementing class implements both the
        // Entity Interface and the Business Interface.
        // This method casts it back to the Business Interface.
        Product product = ( Product )
            entityManager.findEntity( Product.class, _productId );
        ...
        return product;
    }
}
```

Here's a local client of the product service (another service within the container's JVM):

```
public Class AnotherServiceBean
implements  AnotherService,          // for the business method someMethod()
            ServiceBean,             // for setServiceContext()
            ContainerManagedTransaction // to receive a minimal
                                       // transaction controller
{
    public void someMethod()
    {
        ...
        // Acquire a reference to the Product Service
        ServiceManager sm =
            getContext().getManagerFactory().getServiceManager();
        ProductService productService =
            (ProductService) sm.getService( ProductService.class );

        // Call the ProductService to get a reference to the product.
        Product product = productService.getProduct( productId );

        // Is this a reference to an Entity or a DTO?
        assert ( product instanceof Entity ); // true

        // Treat this as an Entity.
        // All updates made to the entity are automatically persisted.
        ProductEntity productEntity = ( ProductEntity ) product;
    }
}
```

Here's a remote client of the product service (another tier outside the container's JVM):

```
public class AnotherClient
{
    public static void main( String[] _args )
    {
        ...
        // Acquire a reference to the Product Service
        ServiceManager = ManagerFactory.getInstance().getServiceManager();
        ProductService productService =
            (ProductService) sm.getService( ProductService.class );

        // Call the ProductService to get a reference to the product.
        Product product = productService.getProduct( productId );

        // Is this a reference to an entity or a DTO?
        assert ( product instanceof DTO ); // true

        // Treat this as a DTO. Updates made to it are purely local.
        ProductDTO productDTO = ( ProductDTO ) product;
    }
}
```

Thus, even though our model hides the difference between remote interfaces and local interfaces for Services and Conversations, a client can tell with a simple check whether it finally gets a reference to a real Entity or just a DTO. So the semantics of local and remote are moved from the Service's interface to the interface of the returned objects themselves.

If the developer has wisely refrained from making the business interface extend the Entity identity interface, this transformation by the framework (to Entity or DTO) becomes extremely simple and elegant, otherwise the DTO received by remote clients will end up having an unnecessary `getPrimaryKey()` method, which will return a null.

What if a DTO is extracted from an Entity and sent to a remote client? Then there are three objects in all, - the Entity, the original DTO and the copy of the DTO which also implements the DTO interface. Clearly, updates made by the original Service on the DTO it extracted from the Entity will not affect the DTO held by the remote client, and vice-versa. Though this looks like a potential problem, the ultimate object the two components are likely to be interested in is the Entity. Both know that what they have is not the Entity. The remote client will only ever try to pass back the DTO to a Service to update the Entity, and the Service will only ever update the Entity by passing the DTO to an Entity Manager and getting it to update the Entity.

If Entities automatically become DTOs when transported to other tiers, is there any value in extracting a DTO explicitly from an Entity? Yes, for two reasons.

Extracting a DTO (whether empty or populated) is an explicit way of creating a template. This template can be populated or modified and then passed back to the Entity Manager to create or update an Entity. The DTO doesn't have to cross tiers or even components. It's a convenient alternative to the multiple createXXX() methods of Entities that take different parameters.

The second reason to extract an explicit DTO from an Entity is to ensure that entity references are **not** passed to other components by mistake, if those components happen to be in the same VM. The DTO helps to make pass-by-value explicit.

DTOs that do not Map to Entities

DTOs that do not map to Entities are actually more common than ones that do. Here too, we suggest some best practices to maximise the value provided by the framework.

Instead of straightaway defining a class containing properties, it is advisable to declare a business interface with those properties and then implement the interface separately.

I.e., it is better to do this:

```
public interface ApplicationForm
{
    public void setName( String _name );
    public String getName();
    public void setAddress( String _address );
    public String getAddress();
    ...
}

public class ApplicationFormImpl
implements ApplicationForm
{
    private String name;
    private String address;
    ...

    public void setName( String _name ) { name = _name; }
    public String getName() { return name; }
    public void setAddress( String _address ) { address = _address; }
    public String getAddress() { return address; }
    ...
}
```

than to do this:

```
public class ApplicationForm // no interface to implement!
{
    private String name;
    private String address;
    ...

    public void setName( String _name ) { name = _name; }
    public String getName() { return name; }
    public void setAddress( String _address ) { address = _address; }
    public String getAddress() { return address; }
    ...
}
```

The advantage of separating the interface from the implementation is that the client can query any object returned by a Service (declared as implementing the business interface) to see if it has a reference to the original one or to a copy, because the framework generates a specially deserialised object for a remote client that implements DTO in addition to the business interface.

```
if ( applicationForm instanceof DTO )
    // We have a copy, not the original
```

[To answer one of Gavin King's criticisms about the business interface concept for data objects, this is not about container concerns (i.e., entity proxying) getting in the face of the developer. It is about managing local-remote differences sensibly. The developer is forced to confront this issue at some point.]

Remote Creation of Objects

Evan Ireland of the EJB 3.0 spec committee asked us how a DTO can be successfully serialised to a remote client in our model, when the remote client typically has no knowledge of the implementing class on the server side.

On the remote client, we have a `ManagerFactory` object, a `ServiceManager` object and a `Service` stub (each has been a factory for the next). A client component has just invoked a method on the `Service` stub that is declared to return the business interface implemented by the `Entity`. The container knows that the client is remote, so it creates a copy of the entity, which implements the `DTO` marker interface and also the same business interface as the entity. This is what it serialises and sends down to the remote client.

Now, the `Service` stub already has access to the business interface definition of the `Entity`. This is bundled with the client application, and is one of the interfaces which the `DTO` implements. But before the `Service` stub can successfully deserialise the object that the container sends over the wire, its class loader (created by the `ManagerFactory`) must request the server for two things: (i) the definition of the implementing class which is available only on the server, and (ii) the serialised object itself. The server serialises the 'class' object and the classloader on the remote client deserialises it. Then when the `Service` stub fetches the serialised object itself, it knows how to reconstitute it because it has already obtained its class definition. It then casts the object to the business interface for a client component to use. Thus, the remote client does not need to have prior access to the value type class which is container-specific and therefore unknown to the client. Since the object also implements the `DTO` interface, the client can detect that it is not the `Entity` but a copy.

A similar mechanism is used to obtain `Service` stubs and `Conversation` stubs themselves from the `ServiceManager` and `ConversationManager` objects.

We recognise that this is the simplest case. As the object graph gets more complex, we will need more sophisticated mechanisms on top of this such as lazy loading, but we believe this is both do-able and worth doing.

A major payoff will be the complete independence of client code from vendor-specific jar files. There will be a common framework jar file that every remote client must bundle, but the client will be completely shielded from changes such as migration to another app server on the server side.

Queries and Bulk Operations

The simplest kind of query is Query By Example. Extract a QueryTemplate from an Entity or its business interface, populate some of its properties and pass it as a parameter to the Entity Manager's findEntities() method to find all entities matching that set of properties. The method returns a Collection (which may be empty if no matches were found). This is the most basic level of query, and allows for optional parameters.

At a slightly higher level of complexity, QueryTemplates allow conditions to be added. These are “and” conditions, and they may specify various expressions, not just equality conditions for properties.

Arbitrarily complex queries can also be written in EJB-QL and represented as abstract methods in a custom entity manager class. Other methods written in Java are written into the bean itself.

If dynamic queries need to be supported to enable search screens, EJB-QL needs to be enhanced with features such as optional parameters and named parameters.

The Process View

We don't believe that Stateless Session Beans and Stateful Session Beans can realistically share the same interfaces and be distinguished by a mere deployment descriptor entry ("Stateful" or "Stateless"). If this was realistic, a developer could write a Session Bean, and a deployer could change the DD entry from "Stateless" to "Stateful" at will, and everything would still work! Clearly, there are more fundamental differences between the two, and if they are not modelled correctly, it will lead to redundant or meaningless methods, as is indeed the case (See Appendix A). This is one of the major flaws with the EJB 2.x model, which the EJB 3.0 spec leaves unaddressed.

One of the fundamental differences in the way our model treats processes, compared to both EJB 2.x and 3.0, is that we have first-class components to represent Stateless and Stateful processes, called Services and Conversations respectively. They each have interfaces and methods that make sense in their own context, and do not attempt to behave like each other.

All processes can act on data objects (Entities, DTOs and QueryTemplates). Business logic is encapsulated within these components (Services and Conversations). Stateless processes (Services) may be triggered by Event Listeners. Entities and Event Listeners have very minimal process logic. Entities may perform some validation, and Event Listeners may perform some data marshalling/unmarshalling between the event and the process they call, but the main process logic (business logic) is always within a Service or a Conversation.

Transactions and state are important aspects of processes.

Transactions – Container-Managed and Bean-Managed

Transactions provide atomicity of operation for processes. A bean may either manage its own transaction or ask the container to manage the transaction on its behalf. The latter option is usually preferable, unless it is imperative to keep the critical section as small as possible (more fine-grained than a method), or to keep a transaction open between requests (very bad for concurrency and not advised unless unavoidable).

In our model, just as with Persistence, it is possible for a bean to tell the container which model it wants, by simply implementing either the ContainerManagedTransaction or BeanManagedTransaction interface. These are marker interfaces that indicate to the container that either a basic transaction controller or a fine-grained one should be placed into the bean's Context.

EJB 2.x defines six transaction types under Container-Managed Transactions. Our model uses the same types, but with more intuitive (if slightly wordier) names.

<i>Transaction Type</i>	<i>EJB 2.x Equivalent</i>
StrictlyTransactional	Mandatory
Transactional	Required
(nothing specified)	Supports
Nontransactional	NotSupported
StrictlyNontransactional	Never
IndependentlyTransactional	RequiresNew

Event Listeners are always CMT, and their transaction types can only be “Transactional” (Required) or “Nontransactional” (NotSupported). If nothing is specified, it means Nontransactional, because no transactional context is passed into an Event Listener.

State

Only Conversations maintain state. Services are stateless. This difference has several implications. The first is that only Conversation components have an identity, while all Services are identical. The second is that only Conversations have a non-trivial lifecycle on both the client and container sides, because state makes them client-specific as well as requiring of management. The third is that while Conversations usually have instance variables, Services must not declare them at all, because they are not thread-safe. The container may choose to use the same bean instance of a Service to multiplex more than one simultaneous call to the same method. So only local variables (which are specific to a single call) and static variables (which are explicitly shared) may be used within Services. While a Conversation is free to declare instance variables, it must also implement public or protected accessors for the ones that it wants to maintain in session scope. Other instance variables are treated as transient.

The Event View

Events are asynchronous and generally not initiated by clients (although we have heard of a proposal for “asynchronous beans”, to be called by clients that don't block on the response).

The two types of events that are handled by EJBs are messages and timer events. The architecture of the two should ideally be the same, but each happens to be implemented differently in EJB 2.x.

In our model, we have a uniform method of dealing with events. We have managed components called Event Listeners, which are stateless and pooled, just like Service Beans. However, they differ from Service Beans in some crucial respects. They are not meant to encapsulate business logic at all. For this, they need to look up and call Service Beans. It is also not necessary to write many business methods for Event Listeners. Event interfaces tend to be standardised and well-known, as opposed to Service business interfaces, which are highly varied and specific to each Service Bean. The variability in Event Listeners comes from the parameters passed to them. So each Event Listener type has a single method corresponding to a type of event. The call-specific parameters are encapsulated in the event object that is passed to the method.

Messages

Our model of a Message Listener may not seem to differ much from that of Message-Driven Beans at one level, yet it is philosophically very different. What the EJB 2.x model refers to as a Message-Driven Bean is a *single* component, whereas we treat the term as referring to a *composite* component made up of an Event Listener and a Service. In usage though, the two tend to be roughly the same. Good EJB design practice emphasises the use of Message-Driven Beans as listeners alone. Business logic is delegated to a Stateless Session Bean.

So in practice, our model is not much different from EJB 2.x, but because we have explicitly identified the need for a listener component as distinct from a component that encapsulates business logic, our model for timers follows the same pattern and is so much superior to the one proposed in EJB 2.1.

Timeouts

We are not sure where to begin to criticise the EJB 2.1 model of timers. Perhaps a description of our model is more appropriate here, with the criticism of Timers moved to Appendix A.

Just like there is a MessageListener in our model, there is a TimeoutListener as well. This is another stateless, managed component. It has a single method in the event interface it implements (“onTimeout()”), and this takes a single parameter (a TimeoutEvent object), similar to the single Message object that is the parameter to the Message Listener's “onMessage()” method. Again, in similar fashion, the body of the method consists only of unmarshalling the parameters that are part of the TimeoutEvent object and using them as parameters to call a Service Bean.

The way timers are *set* is independent of the listener component. Any process bean (Service or Conversation) can set a timer. The bean acquires a TimerManager through the ManagerFactory that it has in its context object, and requests the creation of a Timer object. On the timer, it sets the timeout (through a date, or by specifying the time remaining) and

optionally any recurrence interval. It bundles all required parameters into a TimeoutEvent object and gives it to the timer. Finally, it tells the timer the class of the TimeoutListener that must be called on the timeout.

This way, a calling process can code a “delayed” or “recurring” call to a Service. The Timer, TimeoutEvent object and the TimeoutListener together make it possible for one process to call another asynchronously.

With this model, it is also possible to support “System Timeouts” that are not initiated by any process. These can be configured through an XML file that the server reads on startup and may correspond to batch jobs that need to be run at certain times, independently of client processes.

Things that Don't Belong

Some aspects of the EJB spec make it unnecessarily large, because they really don't belong.

Security

EJB tries to address both authentication and authorisation, although EJB security is predominantly about authorisation.

The EJB tier handles authentication by accessing the context variables “java.naming.security.principal” and “java.naming.security.credentials” that were passed in by the client tier, and passing these in turn to an authentication service to validate.

It handles authorisation by mapping users to roles and roles to resources. The resources in question are EJBs and their methods.

The spec's approach to authentication is perhaps sounder. With authentication, it provides a framework, but none of the implementation. The actual implementation is provided by custom components within the larger organisational architecture of the EJB-based application, such as enterprise directories that hold the actual user identifiers and their authentication tokens. The JAAS API allows for customised authentication components to be plugged in to provide the required functionality. But with Authorisation, the model tries to provide a much more detailed specification of how the implementation should work, which is not a very desirable (or ultimately successful) approach.

Real-world authorisation typically requires access to a Rules Engine with business rules. Much of this authorisation is data-driven. The access control that applications are required to provide is not about access to system components like EJBs, but about access to domain components and concepts like specific bank accounts, groups of products, monetary and other limits, etc. Some of these can of course be successfully proxied by defining access controls on EJBs and their methods, but many others cannot. How can an application ensure that certain bank tellers can only access certain account number ranges, or that a given customer can transact a maximum of X dollars a day, or Y dollars in a single transaction? There is no elegant way to map these concepts to access control rules on EJBs and their methods.

So all too often, application developers end up building a parallel authorisation mechanism to implement the requirements of their application. Which leads to the question, what is authorisation doing in the spec at all?

We believe that rather than go on in great detail about roles and role access to EJBs and methods, the spec should be entirely silent on authorisation. It should just provide a framework for application developers to plug in authorisation modules customised to their organisation's business rules.

The spec will gain immensely by losing a few pounds of fat.

Web Services

Although they seem related, we believe a Service must remain ignorant of Web Services, otherwise encapsulation is badly broken. A Web Service is a layer above a Service, because it represents a specific manifestation of the general stateless service concept, which is tied to a particular content encoding standard (SOAP/XML) and perhaps to a protocol as well (HTTP). If the lower layer (the Service) becomes aware of the higher (the Web Service), encapsulation is broken, which is what has happened in the EJB 2.1 spec.

A Service Bean implements business methods, period. The content encoding and the transport protocol that a Service's client uses is not, and should not be, the concern of the bean. Adapters should be used to abstract these away from the Service.

Our model to support Web Services requires two layers in front of the Service Bean.

The first layer should be a transport interface. For HTTP, the transport interface is the web container, which is not part of EJB at all. For asynchronous Web Services calls that come in on queues, the transport interface is the JMS MessageListener, which is a managed component within the EJB container.

The second layer should be a content/encoding interface. This acts like an adapter to the session bean that translates messages from XML to Java and back by encapsulating a SOAP parser/generator. Both the MessageListener and the Web container need to call the parser before the request goes to the Service. For web calls, the parsing occurs in the web tier. For asynchronous messages, the parsing occurs inside a helper class called by the MessageListener as part of its data unmarshalling.

Once the call is translated from SOAP into Java, the corresponding Service Bean can be called. The Service Bean itself is oblivious to the origin and nature of the call. When the Service Bean responds, the response goes back through the same two layers in reverse order. For HTTP calls, the Service's response is translated by the SOAP generator in the web tier, and a servlet may use the returned data to format its response to its HTTP-based client. For asynchronous messages, the Event Listener must capture the Service's response, call a helper class to marshal the data into a SOAP format, then place the response in an output queue. The Service Bean should remain safely behind these two layers, responding to regular method calls and blissfully ignorant of Web Services. The concept of a "web service endpoint" that is part of a Session Bean's definition in EJB 2.1 is a dependency that should be removed.

[We recognise that not all Service Bean methods should be exposed as Web Services. Obviously, an authorisation layer will have to be negotiated before the bean is accessed, and this is where such constraints belong. The bean itself shouldn't have to know anything about these restrictions.]

Design Artifacts for the New Model

We have three different sets of artifacts to explain our model, and readers are invited to study the ones that best suit their learning style.

1. The UML diagrams show the drawbacks of EJB 2.1 and how our refactored model looks.
2. The code examples show how EJBs under our model may be defined by a developer, and used by client applications.
3. The Javadocs describe the API of our model in some detail.

Our critique of EJB 3.0 follows.

Part IV

Feedback on the EJB 3.0 EDR

We paste sections of interest from the spec document (in italics) and comment on them.

Section 1.1 (Overview)

EJB 3.0 enables the enterprise application developer to program with an API designed for ease of development that is a simplification of the APIs defined by earlier versions of the EJB specification.

[...]

This statement of objectives simultaneously excuses and damns the committee. Given what they have stated they are trying to do, the rest of the document is perhaps understandable. But mere *simplification of the APIs for a developer* is not what EJB 3.0 should be all about. It should address the *simplification of the EJB model* itself! That will automatically simplify the APIs that the developer will see.

[...]

The existing EJB 2.1 APIs remain available for use in applications that require them. The API introduced by EJB 3.0 does not replace or deprecate those earlier APIs.

[...]

The spec should probably only promise that transitional app servers will feature both EJB 2.1 containers and EJB 3.0 containers side-by-side, to provide backwards-compatibility. But why should the EJB 3.0 API not replace or deprecate earlier ones? The spec has passed up a rare opportunity to refactor the model.

Section 1.2 (Goals of this Release)

[...]

Encapsulation of environmental dependencies and JNDI access through the use of annotations, dependency injections, and simple lookup mechanisms.

[...]

These themes keep recurring throughout the spec – the emphasis on both Annotations and Dependency Injection.

Our model shows that the EJB model can be simplified with very minimal use of these features. It also provides simple lookup that encapsulates JNDI. There is an over-reliance on annotations and Dependency Injection in the spec.

[...]

No interfaces are required for Entity Beans.

[...]

We believe there *is* a need for an interface, even for an Entity Bean. This is what has helped us design a consistent data object model that covers Entities, Data Transfer Objects and Query Templates. We have separated out the business interface of a data object (the set of

properties) from what makes the object an Entity, and this separation allows us to re-use the business interface for other data objects such as DTOs. The business interface is too powerful a feature to throw away.

[...]

Elimination of the requirement for the implementation of callback interfaces.

[...]

We think this is a knee-jerk reaction to the problem of empty callback implementations that are such a common phenomenon. The real solution, again, is refactoring of the model so that only relevant callbacks need to be implemented by beans. The “magic” of the container only calling those methods that a bean chooses to implement is both “un-Java” and unnecessary. It may also lead to wrongful use (What if a Stateless Session Bean implements an Entity Bean's `ejbStore()` method? Will the container obligingly call it back? A whole new set of rules will predictably spring up to outlaw such use, all of which are unnecessary if a little commonsense is applied right at the beginning.)

Our model has rationalised the interfaces well enough to reduce the number of methods that a bean has to implement to achieve some functionality. Any reduction beyond this is difficult to explain and test. Remember Einstein's admonition (“Make things as simple as possible, but no simpler”).

[...]

Improved ability for testing outside the container.

[...]

This is a great slogan, but we wonder how achievable it is, especially given the preponderance of annotations, the number of places where dependency injection occurs, and the existence of “magic” callbacks that have no governing interface. It takes a container to understand these. A standalone framework like JUnit would be out of its depth handling 3.0 EJBs, but a JUnit-style framework is what many people think of when they hear “testability outside the container”. The testability that is possible on the POJO part of an EJB (shorn of its annotations and minus the callbacks and injected resources) is likely to be fairly trivial and unsatisfactory.

It may be necessary to set expectations at a reasonable level here. What makes an EJB an EJB is its “managedness”, which means that a lot of its functionality is implemented by the container or defined through callback interfaces. Even its regular business methods are likely to be dependent on values set within callback methods. When a developer tests an EJB, they are implicitly exercising many parts of the container's operation as well, and so EJBs are, almost by definition, not testable outside a container!

Guess what? We think the testing framework that people talk about is likely to be nothing other than a modified EJB container. Yes, if this framework is well-designed, testing can be made easier than it is now, but if someone is hoping for the spec to swoop down and turn EJBs into POJOs, they aren't being realistic.

In fact, our framework lends itself more easily to testability outside the container because of its minimal use of annotations and dependency injection, and its disciplined use of interfaces

to implement callbacks. It is easier to build lightweight frameworks to understand these mechanisms and simulate container behaviour.

Section 2.2 (Business Interfaces)

We disagree with the spec's definition and usage of the term “Business Interface”. The spec claims that Entity Beans do not have business interfaces, but that Message-Driven Beans do (`javax.jms.MessageListener`). Perhaps the spec's authors view a business interface as a collection of methods that do something non-trivial (as opposed to getters and setters).

However, our definition of a business interface is a collection of *domain-specific* methods rather than *domain-neutral* ones. Those methods could be mere accessors, but they nevertheless form a business interface. E.g., the “`setAccountName()`” accessor method of an Account entity is specific to the banking domain, and therefore the “Account” interface that contains that method is a Business Interface. In contrast, the “`onMessage()`” method of `javax.jms.MessageListener` is not domain-specific. It is an “Event Interface” rather than a “Business Interface”.

We are not trying to split hairs here. Business interfaces are *essential* for Entities, Services and Conversations. It is not just a container concern that is related to proxying. For data objects like Entities, it is something that the developer has to handle anyway in terms of distinguishing instances of pass-by-value from instances of pass-by-reference. Having a distinct business interface for Entities allows us to model Data Transfer Objects elegantly. Its importance can be judged from the fact that the EJB 3.0 spec, having forsworn the use of business interfaces for entities, is forced to implement DTOs by actually detaching entities themselves!

The spec also wastes too much time and space on the possibility of generating interfaces from beans, and ways of doing so. We think the spec should remain loftily silent on this. All that the spec should demand is that by the time the application is deployed into a container, the business interface corresponding to the bean must exist as a separate file. Whether the developer hand-codes both files, or generates one from the other (using annotations or a tool like XDoclet) should not be the concern of the spec. Who will be responsible for generating the interface from the bean's annotations, anyway? The container? Then errors can only be detected at deployment time and not before. We can't help pointing out that this is another factor that militates against the objective of “testability outside the container”. It starts getting too unnecessarily tangled, and we would like to see this discussion dropped from the spec. It's an implementation detail that doesn't belong.

Chapter 3 (Stateless Session Beans)

Section 3.1.1 (Business Interface)

This is more properly a rant against EJB 2.1, but the EJB 3.0 spec defines an annotation called “`WebMethod`”, without addressing the fundamental design flaw in the EJB 2.1 spec, and so the rant's place here is justified.

Please keep Session Beans ignorant of Web Services, otherwise it badly breaks encapsulation! (A read through the EJB 2.1 spec shows that the “web service endpoint interface” is not layered above the regular service interface of the Session Bean, but at the same level. The Bean is aware of the interface that it supports, and even has a `getMessageContext()` method in its context to obtain information about the Web Service call. Web Services tags also invade the Session Bean's deployment descriptor, but this is a lesser evil.)

We would like to see Web Services *layered* on top of Services without the Services being aware of them. That is true encapsulation. If a new technology called XYZ Services appears tomorrow, will you modify your beans once more, or just create a new wrapper to handle those services?

We don't object to the `WebMethod` annotation *per se*, but it should not be implemented without fixing the underlying model. We believe that the annotation should be an indication to the container to transparently create wrapper layers on top of Services, nothing more.

As long as the underlying model is not fixed, annotations are just icing on mud.

Section 3.1.3 (Bean Class)

[...]

A stateless session bean must be annotated with the `Stateless` annotation or must implement the `javax.ejb.SessionBean` interface.

[...]

There is a more fundamental problem here – that Stateful and Stateless session beans are very different components and should not be cavalierly distinguished by a mere deployment descriptor entry. The spec needs to recognise this and propose first-class component names for the two, such as the ones we have proposed (Service and Conversation). The need for an annotation will melt away when the component model is rationalised.

[Gavin King of the Hibernate project and a member of the EJB 3.0 spec committee thinks there is nothing cavalier about treating Stateless and Stateful Session Beans alike. If he's right, it should be possible to change a Session Bean's behaviour from Stateful to Stateless just by changing the deployment descriptor entry!]

[...]

A stateless session bean that is annotated with the `Stateless` annotation may (but is not required to) implement the `setSessionContext` method of the `javax.ejb.SessionBean` interface.

[...]

The bean class may optionally implement the `ejbCreate` and/or `ejbRemove` method. If it does, the container will call these methods as required by the EJB 2.1 specification.

[...]

We're very uncomfortable with the introduction of new rules in EJB that relax the discipline of existing Java interface law. Under Java law, if a component claims to implement an interface, it must implement all methods in that interface. Yes, it can implement *similarly-named* methods of its own without implementing the interface, but then there is no implied

relationship between those methods and the ones in the interface. The spec is saying that if a component implements a method called “setSessionContext()” without implementing the interface that contains it, the container will still magically associate that method with the method of the same name in the interface!

Consider this: If a bean contains an innocently-named method “setXYZ()” that doesn't correspond to any callback today, but a future version of the spec declares this a callback method, then the application will break, because the container will assume this method is the callback method and start calling it, even though the bean doesn't implement any callback interface!

[Gavin King claims there is no magic in the callbacks. We disagree. The “magic” occurs when the container *assumes* that a method implemented by a bean is actually the same as a similarly-named method in some interface that the bean does not even implement!]

Needless to say, we think this provision is unwarranted and even dangerous. We need the discipline of an interface to guard against situations like this. The container must not assume that arbitrary methods are callbacks when the corresponding interface is not implemented.

In our model, the callback methods have been rationalised into sensible interfaces with a limited number of related methods in each. In software engineering terms, we have increased cohesion (by grouping only related ones within an interface) and decreased coupling (by separating unrelated ones into different interfaces). In addition, many of the interfaces are now optional or mere markers. We believe we have alleviated the developer's pain and made development effort reasonable. Beans with simple functionality can ignore the optional interfaces, while those with more complex functionality must only implement relevant interfaces and the few methods in them.

Our approach is boring, but safe, correct and justifiable. The spec's approach is cool, but dangerous and unjustified.

Section 3.1.4 (Dependency Injection)

If a stateless session bean uses dependency injection mechanisms for the acquisition of resources or other objects in its environment (see Chapter 8), the container injects these references when the stateless session bean is transitioned to the “method-ready” state.
[...]

See our earlier discussion on Dependency Injection and Service Locator, and how our model has achieved the best of both worlds by placing all Service Locator references within a single injected context object.

[Evan Ireland of the EJB 3.0 spec committee pointed out that resource dependency annotations have an advantage over Service Locators in that the container or deployer can know about resource dependencies at deployment time.

Our point is that this can only work with 100% confidence if Service Locators are completely outlawed, because Service Locators express dependencies programmatically, and these can be very difficult or impossible to deduce at deployment time. Since Service Locators are too useful to be outlawed (especially simple getter-based Locators like the ones in our model), we can probably never achieve complete deployment time knowledge of all resource dependencies, so the objective is probably unrealistic anyway.]

Section 3.1.5 (Client View)

The local or remote client of a session bean acquires a reference to a session bean business interface either through one of the dependency injection or lookup mechanisms described in Chapter 8. Alternatively, the client may use the explicit JNDI lookup mechanisms described by the EJB 2.1 specification.

There is no need for a plethora of such mechanisms. Client-side component lifecycle methods are standard for stateful and stateless session beans. They are part of the framework as Manager classes that transparently look up and retrieve a stub for the client.

i.e., instead of

JNDI -> Home interface -> Component interface

as in the EJB 2.x model, we do

Manager -> Business interface

in our proposed model.

Example:

```
// Get the manager (home interface):
ServiceManager serviceManager =
    context.getManagerFactory().getServiceManager();

// Get the stub that implements the business interface.
// The client doesn't have to care if the service is local or remote.
MyService myService = (MyService)
    serviceManager.getService( MyService.class );

// Use the stub as if it's the bean.
// The business interface hides the difference.
myService.someBusinessMethod();
```

Also see our code examples.

Chapter 4 (Stateful Session Beans)

Section 4.1.2 (Home Interface)

Stateful session beans do not require home interfaces. Under the EJB 3.0 API, home interfaces are typically not used for stateful session beans.

This is another statement we cannot understand or agree with. Home interfaces (whether they are called that or something else) encapsulate client-side lifecycle operations. Don't stateful session beans require client-side lifecycle management? How will a client initiate a session? How will they signal that they're done with a session? If a method has to re-acquire a session from another client device using a serialised handle, how would that be done without a home interface?

We have the equivalent of a home interface in our model. We call it ConversationManager. It has methods such as “initiateConversation()”, “findConversation(Handle h)” and endConversation(Conversation c)”, which make it extremely simple for clients to manage Conversations.

Section 4.1.4 (Dependency Injection)

See comments under 1.2.

Section 4.1.5 (Client View)

[...]

When stateful session bean (sic) is injected into a client context or is obtained by lookup, the container creates a new stateful session bean instance to which method invocations from the client are delegated. This instance, however, is uninitialised from the client's point of view, since as (sic) the client does not call an explicit “create” method to obtain and initialise the bean.

[...]

Something puzzles us enormously here. This is a *client* component we're talking about. It could exist on another tier altogether, outside the container, perhaps as a standalone J2SE application. *Who injects the session bean into it?*

That apart, the spec is placing responsibility for things that are “above” the level of a bean on the bean itself – a classic EJB 2.x approach. In other words, what is injected into the client is not the bean itself, but something that works like a home interface. Unsurprisingly, this object has to be explicitly requested for a reference to the actual initialised bean. Why don't they just say they're injecting the home interface? This is clearly not the bean, because the bean can only be acquired by requesting this injected object (whatever it is) for it.

Our model is straightforward and far less confusing.

A client looks up a ConversationManager, which in turn gives it a reference to a newly-created Conversation. It can then start using the Conversation because it is initialised. How does the client get access to the ConversationManager in the first place? Through a ManagerFactory. How does it find the ManagerFactory, then? This is a class bundled with the client application in a standard jar file that all clients of the EJB tier must have in their classpath.

It works as follows:

```
// xyz.client.ManagerFactory is in the client application's classpath.
ConversationManager cm = ManagerFactory.getInstance()
    .getConversationManager();

// The Conversation is explicitly requested, and is already initialised when
// the client receives a reference to it.
MyConversation myConversation = (MyConversation)
    cm.initiateConversation( MyConversation.class );

// Use the Conversation.
MyConversation.someBusinessMethod();
```

It's usually only non-EJB clients (associated with human users) that call Conversations. We can't think of a case where a Service or another Conversation would act like a client to a Conversation, but if there is such a requirement, our model can support it trivially. In fact, for in-container clients like Services and Conversations, our model is exactly the same as what the spec proposes, because all EJBs have an injected context from where they can find home

interfaces (Managers). The managers have to be explicitly requested for a reference to a fully-initialised Conversation, as always.

The first statement in the previous example is now replaced with

```
ConversationManager cm = context.getManagerFactory().getConversationManager();
```

and all subsequent steps are as before.

This is another example of the spec's over-reliance on dependency injection. It still doesn't make life simple for the developer, because the developer still has to explicitly initialise the bean. Additionally, disguising the home interface as the bean at injection time and claiming there is no home interface seems dishonest!

Section 4.1.6 (Stateful Session Bean Removal)

The Remove annotation may be used to annotate a stateful session bean method. Use of this annotation will cause the container to remove the stateful session bean instance after the completion (normal or abnormal) of the annotated method.

[...]

This is a really interesting and original proposal. It seems to be much more robust than requiring the client to be nice and call “remove()” on the bean when they're done. Frankly, we have no equivalent to this in our model, except a programmatic one.

i.e, at the end of such a method, the bean should ask the Manager class to remove it:

```
// I'm done now. Remove me, O Manager!  
getConversationContext().getManagerFactory().getConversationManager()  
    .endConversation( getConversationContext().getThis() );
```

Where a convenience class is used, the syntax looks a bit less forbidding:

```
getManagerFactory().getConversationManager()  
    .endConversation( getThis() );
```

Yes, we could do it, but the Remove annotation seems much more elegant. In keeping with the naming convention we use for the client-side lifecycle method (“endConversation()”) and the container-side lifecycle callback (“beforeEnd()”), we would however call the annotation “EndsConversation”.

Chapter 5 (Message-Driven Beans)

Section 5.1.1 (Business Interface)

The business interface of a message-driven bean is the message-listener interface that is

determined by the messaging type in use for the bean. For example, in the case of JMS, this is the `javax.jms.MessageListener` interface.

[...]

As we said before, we believe that this is an “Event Interface” and not a “Business Interface”. If the timer mechanism had been implemented similarly (sigh), it would have been so much more elegant, but that's another EJB 2.1 rant.

Section 5.1.3 (Dependency Injection)

See comments under 1.2.

Section 6.1 (Requirements on the Entity Bean Class)

[...]

An entity may allow its persistent state to be accessed by the container either via the JavaBeans style property accessors or protected instance variables. If the entity is annotated with the annotation member value `access=PROPERTY`, or if the access annotation member value is not specified, the container accesses persistent state via the property accessor methods and all properties not annotated with the `Transient` annotation are persistent.

If the entity is annotated with the annotation member value `access=FIELD`, the container accesses instance variables directly and all non-transient instance variables that are not annotated with the `Transient` annotation are persistent. The persistent fields of the bean class must be protected.

[...]

Our emphasis on business interfaces for Entity Beans makes it necessary to think in terms of properties rather than fields.

The spec has also chosen a policy of “annotation by exception”, which is why the annotation is “`Transient`” rather than “`Persistent`”. In our model, properties not explicitly mapped to a column (by whatever mechanism) are automatically transient. We have not discussed persistence mapping in this document, because our focus is on rationalising the component model.

Section 6.1.7 (Inheritance)

A lot of emphasis is placed on mapping class hierarchies to tables. It's good that the spec is trying to bring about a vendor-neutral and framework-neutral way to do O-R mapping. Our only concern is that this is a mammoth exercise, and will consume hundreds of person-hours of work that could be better utilised on fixing more urgent problems.

Section 6.2 (Entity Manager)

Section 6.2.1 (Entity Manager Interface)

The Entity Manager described in the spec is conceptually similar to the one we propose. However, there are some important differences.

- The spec conceptualises “merge” and “detach” as fundamental operations, and even defines corresponding lifecycle callback methods (`ejbDetach()` and `ejbMerge()`). The container-side lifecycle is now more complex, because of the larger number of states and methods. The Entity Manager's client-side interface has methods called “flush()”, “refresh()” and “evict()”, reflecting these state transitions.

Our model does not tamper with Entities and their lifecycles to implement Data Transfer Objects. We take snapshots of Entities instead.

The model proposed by the spec is broken, because:

1. The stated design objective of exposing the domain model to other tiers is antithetical to good software engineering practice (tight coupling as opposed to decoupling of tiers).
2. It confuses DTOs with Entities based on superficial similarities between some DTOs and some Entities in terms of their properties. DTOs are not Entities.
3. It burdens the container-side lifecycle of an entity with aspects local to the business logic (why should taking a snapshot of an entity's state affect its lifecycle?). This is another example of tight coupling.
4. It puts the responsibility of driving these container-side state transitions back on the business logic (e.g., “evict()”). This is yet another example of tight coupling that also breaks the Entity Bean contract (which says that business logic should be kept unaware of persistence concerns).

In short, the whole “detached entity” concept is too fundamentally flawed to be salvaged.

If the application case for detached entities was just to use them as DTOs, then what we have proposed is a simpler and cleaner alternative. However, if detached entities are visualised as a way of “locking” entities on read so they can't be updated by another user, then the correct approach would be to implement isolation levels for entities. Set the isolation level of the entity to `REPEATABLE_READ`, then use a Conversation (Stateful Session Bean) implementing `BeanManagedTransaction` to start a transaction, extract a snapshot (there, you've read the entity, so now the container won't let anyone else update it till your transaction completes) and return the snapshot to the user tier without ending the transaction. When the modified DTO comes back (after the user has had a coffee and everything), update the entity with the values in the DTO, then end the transaction in the second method. Obviously, this is awful for concurrency, but if it's an application requirement, this might be a good way to do it. [We can see where the proposal for a “persistence context that spans transactions” may have come from.]

So, if that was the thinking, we would nudge the spec in the direction of isolation levels for entities and away from detached entities and persistence contexts that span transactions.

- The spec has generalised methods to run arbitrary EJB-QL queries through the Entity Manager. This can be quite complex, and requires the coding of EJB-QL expressions as strings in Java code, leading to an impedance mismatch. In our model, there are several methods to run queries. The simplest, based on query-by-example, uses a QueryTemplate object that has the same business interface as the Entity, and some additional methods to add query conditions. This is ideal for search screens that refer to a single entity. For more complex queries, the interface EntityManager can be subclassed by interfaces containing custom finders, bulk updates and bulk deletes. These are implemented in EJB-QL and placed in deployment descriptors. For a client, calling an intuitively-named Java method (e.g., findByPriceRange(double, double)) is more natural than composing a query string and firing it through a general-purpose query mechanism.

One point that Gavin King inadvertently highlighted by his mention of search screens and dynamic queries is the need for optional parameters in EJB-QL.

Section 6.2.2 (Example of Use of Entity Manager API)

```
[...]
@Inject public void setEntityManager( EntityManager em ) {
    this.em = em;
}
[...]
```

This is unnecessary in our model. The Entity Manager is readily accessible through the bean's context.

```
EntityManager em = context.getManagerFactory().getEntityManager();
```

Section 6.3 (Entity Bean Life Cycle)

An entity bean instance is in one of four possible states: new, managed, detached and removed.

[...]

We don't think new and detached are valid states. We exploit DTOs to provide the same functionality in a simpler and cleaner way.

[...]

A new entity bean instance has no persistent identity, and is not yet associated with a persistence context.

A managed entity bean instance is an instance with a persistent identity, currently associated with a persistence context.

[...]

As we explained earlier, we don't need persistence contexts. If the spec is trying to keep

entities detached across transactions to implement readlocks, then isolation levels might be the answer (e.g., REPEATABLE_READ).

Entities are *always* managed by the container, *all the time*. There are no such things as new or detached entities.

This is our equivalent of the spec's "new" entity:

```
// Get an empty DTO from the business interface definition.  
MyDTO myDTO =  
    ( MyDTO ) entityManager.getDTO( MyEntity.class );
```

"MyDTO" is the DTO that implements the same business interface implemented by "MyEntity".

Obviously, "myDTO" is not managed. The container doesn't even know about it. The EntityManager has created a Data Transfer Object on the client side based on its business interface. You can populate this with any combination of attributes and use it to create an entity. There is no need for a variety of createXXX() methods. The EntityManager will make appropriate calls to the container to have it created, then return you a reference to the newly-created, *managed* entity.

This is an example of a DTO being used to *create* an entity:

```
myDTO.setXXX( "junk" );
myDTO.setYYY( "more junk" );
MyEntity myEntity = ( MyEntity ) entityManager.createEntity( myDTO );
```

Here's an example of a DTO being used to *update* an entity:

```
// A snapshot of an entity is obtained as a DTO.
// The entity itself remains managed.
MyDTO myDTO = ( MyDTO ) entityManager.getDTO( myEntity );
```

```
Object primaryKey = myEntity.getPrimaryKey();
```

(Pass the DTO to another tier, say the web tier)

```
// What the other tier receives may either be a reference
// to the same DTO (if in the same VM),
// or a fresh copy of the serialised DTO that also implements
// the DTO interface (if in another VM).
// Either way, the client knows it is not the Entity.
// Updates happen to the DTO in this tier.
// The entity in the EJB tier is obviously unchanged.
myDTO.setXXX( "junk" );
```

(Pass the modified DTO back to the EJB tier)

```
// The updates are now applied from the value object to the
// original entity and a reference to the entity is returned
myEntity = ( MyEntity ) entityManager.updateEntity(
    primaryKey, myDTO );
```

In this example, “myEntity” is the Entity, “myDTO” is the Data Transfer Object. You could squint and look at “myDTO” as a detached entity, because that's what it appears to be on the client side.

But if you take it all very literally and start corrupting the container-side lifecycle with new and unnecessary concepts like “detach”, “merge” and “persistence context”, things start getting hairy. Don't do it.

Sections 6.3.1 to 6.3.4 (Creation, Removal, Synchronisation with the Database, Detached Entities)

These are four absolutely unnecessary sections in the light of what we have discussed above. Keep “detach” and “merge” on the client side where it belongs, and none of the issues discussed in these sections will apply.

Section 6.3.5 (Transaction Rollback)

Transaction rollback causes a newly created instance to revert to the new state. It causes a managed instance to become detached. Portable applications should not try to reuse such instances.

Again, this is an absolutely unnecessary discussion. It certainly sounds complex and important to bear in mind, but do we really need to spell out these situations? Look at our equivalent code:

```
(transaction begin)

// Get an empty template from the class definition:
MyDTO myDTO =
    ( MyDTO ) entityManager.getDTO( MyEntity.class );

// Set properties on the template:
myDTO.setXXX( “junk” );

// Create a new entity out of the populated template:
MyEntity myEntity =
    (MyEntity) entityManager.createEntity( myDTO );

(transaction rollback)
```

What do you think happens to the entity? What happens to the DTO?

The only container-side event is the entity creation, and that gets rolled back. So “myEntity” should now have a null or meaningless reference, while “myDTO” continues to cheerfully hold onto its values. It's a client-side object, after all, unaffected by transactions.

So what the spec is trying to say in its laborious way is that DTOs are not affected by transaction rollbacks, but the creation and update of entities are. Which should be pretty obvious.

[We hope we haven't confused anyone with our use of the term “client-side”. The client for an Entity Bean is a Service Bean or Conversation Bean, so the “client-side” is still in the container. But the container doesn't manage data objects that don't implement the Entity identity interface, so DTOs are unmanaged. See the fourth UML diagram pertaining to Entities and DTOs for a more detailed view of the differences.]

Section 6.4 (Extended Persistence Context)

A persistence context that spans multiple transactions is under consideration for a future draft.

Please don't consider the “persistence context” concept anymore! Let it not span more than one draft ;-). We have hopefully demonstrated its utter uselessness as well as its power to confuse and complicate.

Section 6.5 (Callback Methods)

We plan to supply a callback interface for life cycle events. This callback could be implemented by any class, not necessarily the entity bean class.

[...]

There are two issues here.

One relates to the point we made before – if a bean needs to implement a callback method, it must implement the interface that contains it. We think the implementation of stray callback methods is fraught with danger.

The second issue relates to who can implement an interface that is meant for EJBs. We are not being elitist here, but we see a Pandora's box being opened if proletarian classes (pun intended) are allowed to implement callback interfaces meant for EJBs.

We can see that certain non-EJB classes such as Data Access Objects (DAOs) may legitimately want to receive entity lifecycle callbacks, and act as interceptors to BMP Entity Beans.

CMP Entity Beans in the EJB world are far superior to Interceptors in frameworks like Hibernate, because writing nothing beats writing an interceptor! CMP Entity Beans don't have to receive callbacks at all in our model.

So we see a dwindling role for BMP Entity Beans overall. Still, BMP Entity Beans may be unavoidable when Entities wrap systems of record on legacy systems and non-relational datastores.

It may be worthwhile exploring a new managed component (a DAO) that acts like a Decorator for an Entity Bean and manages its persistence mechanism (the “how”) in response to container callbacks that it receives (the “when”). But we don't think arbitrary Java classes should be able to implement EJB callback interfaces and receive callbacks from the container.

[...]

An entity bean may optionally implement any of the following callback methods: `ejbCreate`, `ejbLoad`, `ejbStore`, `ejbRemove`, `ejbMerge`, `ejbDetach`.

The semantics of these methods and their place in the entity bean lifecycle will be defined in a future draft.

[...]

`ejbMerge` and `ejbDetach` do not belong in an Entity's lifecycle. The Hibernate model of the world is different. Business logic takes *responsibility* for POJO persistence (the “when”), while the Hibernate framework provides the *mechanism* (the “how”). It may make sense under that model for business logic to tell Hibernate to lay off a POJO for a while, because Hibernate never takes *responsibility* for persistence anyway.

But Entity Beans (and EJBs in general) differ from POJOs precisely because they are *managed components*. The container and the Entity Bean together take responsibility for persistence. Business logic (i.e., a Service or Conversation) is relieved from the responsibility of persistence (It cares about neither “when” nor “how” an Entity is persisted). In the CMP (Container-Managed Persistence) model, the container takes full responsibility, performing the actual persistence-related operations (the “how”), and doing them whenever required (the “when”). Even the Entity Bean itself never has to know. In the BMP (Bean-Managed Persistence) model, the container only tells the bean about persistence-related events through callbacks (the “when”), and the bean performs the actual persistence logic itself (the “how”). So in the EJB world, unlike in the traditional Hibernate world, it makes no sense for an EJB to stop being managed. Who will take responsibility for persistence then? The concept of a container not managing an EJB is simply heretical, because it violates the Entity Bean contract that promises to keep business logic ignorant of persistence.

If a developer is masochistic enough to code part of persistence logic (the “when”) into business logic (a Service or Conversation), and have the data POJO persisted through Hibernate (the “how”), they are certainly free to do so. But the data POJO in question is *not* an Entity Bean.

We are emphatic about this: Merge and Detach are client-side operations. They are not part of an entity bean's container-side lifecycle at all.

All the others are legitimate callbacks. But what happened to `ejbPostCreate`, `ejbPassivate` and `ejbActivate`? We miss them, and we have created nicely-named equivalents for all of them in our model. See the UML.

Section 6.7 (Security)

We plan to add permissions definitions for the “CRUD” operations (create, read, update, delete) and define a callback mechanism for entity security.

Please don't! Cut security out of the EJB spec altogether. The “component-as-surrogate” model has not worked well at all for domain-oriented application developers for whom authorisation means how much money a customer is allowed to transfer in a single transaction, and not whether a role can access a bean and its methods. Authorisation is something the EJB spec does really badly. It's time to put a merciful end to this clumsy mechanism.

We probably need a new JSR to define authentication and authorisation interceptor plugins that the application developer writes. Corporate directories and rules engines must be leveraged to provide these security constraints. The EJB spec is hopelessly out of its depth in attempting to deal with real-world security.

Section 6.8 (Query API)

The case for this mechanism is much weakened if we refactor the old home interfaces with their custom finders and home business methods. Once we recognise that such code must not find its way into an individual bean in any form, then our design becomes much simpler. Obviously, all such bulk operations belong with an EntityManager. But the standard EntityManager has a limited number of methods.

The spec's approach is to endow the EntityManager with a generic mechanism to execute different kinds of queries, and to therefore have a complex mechanism to define such queries.

For simple queries relating to a single entity, we use a QueryTemplate object extracted from the business interface of the Entity.

```
// Get the EntityManager.
EntityManager em = context.getEntityManagerFactory().getEntityManager();

// Get the query template corresponding to the entity:
MyTemplate myTemplate = (MyTemplate) em.getQueryTemplate( MyEntity.class );

// Populate the query template with selected equality conditions (AND-ed together).
// Other properties should be ignored when the query is executed. The QueryTemplate
// and EntityManager understand which properties should be ignored, but EJB-QL
// needs to support a corresponding syntax.
myTemplate.setXXX( "value 1" );
myTemplate.setYYY( "value 2" );

// Some non-equality conditions may also be added:
myTemplate.addCondition( "someProperty",
                        "QueryTemplate.LESS_THAN",
                        "50.0" );

// The entity manager executes the query and returns a Collection of matching
// entities.
Collection entities = em.findEntities( myTemplate );

/*
// If no custom query template class (MyTemplate) exists, cast the object to the
// business interface or to the identity interface when using the respective methods.
MyData myData = (MyData) em.getQueryTemplate( MyEntity.class );
...
((QueryTemplate)myData).addCondition(...);
...
Collection entities = em.findEntities( (QueryTemplate) myData );
*/
```

Our approach for more complex queries is to refactor the 2.x model. There, the developer had to code all such methods into the respective entity bean class, even though they applied to *a set of entities*.

We modify this technique slightly in our model, by coding such methods into custom subclasses of EntityManager.

The client will call these methods directly, without passing in any query strings.

```
// Get a custom Entity Manager (MyEntityManager),
// not the framework's standard one (EntityManager):
MyEntityManager myEntityManager =
    (MyEntityManager) context.getManagerFactory()
        .getEntityManager( MyEntityManager.class );

// The query is encapsulated within the manager and doesn't have to
// be passed in either as a string or referenced by name.
Collection entities = myEntityManager.findByPriceRange( 10.0, 20.0 );
```

The custom entity manager is defined as follows:

```
public interface MyEntityManager extends EntityManager
{
    public Collection findByPriceRange( double _fromPrice, double _toPrice );
}
```

and implemented as follows:

```
<!-- File "ejb-canned-queries.xml"-->
<canned-queries>
    <query>
        <class-name>MyEntityManager</class-name>

        <method-signature>
            java.util.Collection findByPriceRange( double, double )
        </method-signature>

        <query-string>
            select Object (p) from Product p
            where p.price >= ?1 and p.price <= ?2
        </query-string>
    </query>
</canned-queries>
```

If dynamic queries are required, EJB-QL must be enhanced to support optional parameters.

Chapter 7 (Query Language)

We have been very hard on the EJB spec committee so far. But on this chapter, we stand up and applaud.

EJB-QL badly needs all the enhancements described in this chapter. More power to EJB-QL!

In our model, a more powerful EJB-QL will benefit custom entity managers as shown in the last example.

Chapter 8 (Entity Bean Context and Environment)

It seems a bit offhand to dismiss such a heavy chapter in just a few paragraphs, but we have very little to add to what we have said before about annotations.

Yes, we're dazzled by the number and types of annotations that are possible, but we're still not convinced that all of them are necessary. Our mantra remains the refactoring of the component model.

We also think that Service Locator is a far more straightforward mechanism to use. We can't see a need for Dependency Injection beyond setting the bean's context.

Section 8.1.5 Universal Context

We have an idea that we call “Universal Context”, but it is very different from what the spec proposes. The spec envisages a context object that subsumes the functionality of the earlier EJBContext and the bean's JNDI context. Our model doesn't need such a mechanism, because a bean's Context already gives it access to any resource it may require, through built-in Service Locators.

Our concern was more about non-EJB classes that run within the container (e.g., helper classes to EJBs) and that may require access to EJBs as well as other resources. Being ordinary Java classes and not EJBs, they have no Context object from which they can access a Service Locator.

We see the possibility of providing a UniversalContext object that serves the same purpose for non-EJB classes that the regular Context objects serve for EJBs.

i.e., a non-EJB class would access resources as follows:

```
// Call the UniversalContext's static method "getManagerFactory()":
ResourceManager rm =
    UniversalContext.getManagerFactory().getResourceManager();

DataSource ds = rm.getDataSource( "Production DB" );
```

However, we hesitate to make this a full-fledged part of our model because of its potential for abuse.

Chapter 9 (Metadata Annotations)

When the model is cleaned up through refactoring, half the annotations in this section will no longer be required. In general, the only annotations that are useful are those that reference deployment descriptor entries. Annotations that reference the component model itself are a “smell” that suggests the need to refactor the model.

Chapter 10 (Metadata for Object/Relational Mapping)

In general, we support the idea of framework-neutral or vendor-neutral O-R mapping. But we are filled with unease on seeing the hairy syntax of some of these annotations.

Conclusion

We have presented here an alternative way to simplify EJB for developers by attacking the root of the complexity problem – the component model.

With a refactored component model, EJB technology becomes refreshingly understandable, and its power more readily obvious.

There is a lot more work to be done. The deployment descriptors need to be rationalised, and annotations may be a good way to hide the “impedence mismatch” between Java and XML. But this would be a thoughtful use of annotations, not an indiscriminate one as the current proposal seems to be.

As our proposed model is discussed in greater detail by the Java community at large, we fully expect that many of our own design assumptions and decisions may be challenged. We have already been through one round of feedback from members of the EJB 3.0 spec committee, and this has helped improve our model. We welcome more such feedback.

Is our proposed model backwards-compatible with EJB 2.x? Frankly, we have been more concerned with the correctness and elegance of the component model, and have deliberately ignored backwards-compatibility issues. We'll leave it to those with skin in the game to figure out a way to make this model work with the old one!

Finally, although we have been very critical in this document of the EJB spec committee (past and present), we are nevertheless indebted to them for their tremendous accomplishments and sheer effort. It has been easy for us to criticise what they have produced (“Any fool can criticise, and most fools do”), but we are not sure we would have done much better had we been in their place.

On that note of appreciation and humility, we turn our model over to the community for review and discussion.

Rajat Taneja
Ganesh Prasad

Part V

Code examples

Here are some examples showing how our model may be used to develop multi-tier applications. We have deliberately omitted exception declarations and handling to avoid visual clutter. The package “xyz” is our shorthand for the EJB package, which could become “javax.ejb” if this model is officially adopted.

Sample 1 – Creating an Entity Bean

Entity Beans must implement two interfaces, - an Identity Interface (which is xyz.Entity and which declares a method called “getPrimaryKey()”) and a Business Interface (which is specific to that Entity, and defines its unique set of properties in the form of getters and setters). But we would like all these methods (accessors as well as getPrimaryKey() to be available through a single interface, otherwise clients will have to keep casting the entity back and forth to one of these two interfaces to access the two sets of methods. So we need to define a single “Entity Interface” that trivially extends both the Business Interface and the Entity Identity Interface “xyz.Entity”.

It may be tempting to write the Business Interface to extend the Entity Identity Interface directly, but this can create numerous problems with DTOs (Data Transfer Objects) that also implement the same Business Interface but a different Identity Interface (xyz.DTO).

This certainly seems like a lot of work!

However, we use a convenient shortcut that helps a developer avoid creating two interface files per Entity. Only one file is required, although two interfaces are actually created.

Step 1-1 – Declaring the Business Interface

```
/* Product.java */
package myApp;

public interface Product
{
    public String getId();
    public void setId( String _id );
    public String getDescription();
    public void setDescription( String _desc );
    public double getPrice();
    public void setPrice( double _price );
}
```

Nothing could be simpler. The Business Interface only defines the properties.

Step 1-2 – Declaring the Entity Interface

The Entity interface has to extend the business interface (for the accessors) as well as the Entity Identity Interface (for “getPrimaryKey()”).

This can be done in two ways. We will show the simple and straightforward option first, and then show how to exploit a little-known feature of Java 1.1 to save ourselves some typing.

Option 1:

```
/* ProductEntity.java */
package myApp;
import xyz.Entity;

public interface ProductEntity
extends      Product,
            Entity
{
}
```

Option 2: The easier option is to make the Entity Interface a static inner interface of the Business Interface:

```
/* Product.java */
package myApp;

public interface Product
{
    // Accessors:
    public String getId();
    public void setId( String _id );
    public String getDescription();
    public void setDescription( String _desc );
    public double getPrice();
    public void setPrice( double _price );

    // Use a static inner interface instead of creating a new interface file:
    public interface Entity extends Product, xyz.Entity {}
}
```

The effect of both options is the same. The only syntactic difference is that Entity Bean implementations must implement “ProductEntity” in the first case, and “Product.Entity” in the second. The static inner interface saves typing a file, and is mere syntactic sugar. We will use the shortcut notation throughout these examples.

Step 1-3 – Implementing the Entity Interface

```
/* ProductEntityBean.java */
package myApp;

import xyz.CMPEntityBean;

public class ProductEntityBean
    extends      CMPEntityBean      // Convenience class
                                // (or extend your own superclass
                                // and implement EntityBean and
                                // ContainerManagedPersistence)
    implements   Product.Entity     // Business interface (static
                                // inner interface)
{
    // Attributes:
    private String id;
    private String desc;
    private double price;

    // Accessors (inherited from Business Interface “myApp.Product”
    // through Product.Entity):
    public String getId() { return id ; }
    public void setId( String _id ) { id = _id; }
    public String getDescription() { return desc; }
    public void setDescription( String _desc ) { desc = _desc; }
    public double getPrice() { return price; }
    public void setPrice( double _price ) { price = _price; }

    // Identity method (inherited from Identity Interface “xyz.Entity”
    // through Product.Entity):
    public Object getPrimaryKey() { return getId(); }
}
```

Sample 2 – Creating a Service (Local client to the Entity)

Step 2-1 – Declaring the Business Interface

Here's a Stateless Session Bean (we call it a Service) that operates on the Entity Bean:

```
/* ProductQueryService.java */
package myApp;

import xyz.Service;

public interface ProductQueryService
extends      Service      // Component interface for all Services
{
    public double getPrice( String _productId );
}
```

Step 2-2 – Implementing the Business Interface

```
/* ProductQueryServiceBean.java */
package myApp;

import xyz.CMTServiceBean;
import xyz.ManagerFactory;
import xyz.EntityManager;

public class ProductQueryServiceBean
extends      CMTServiceBean    // Convenience class
                                // (or extend your own superclass
                                // and implement ServiceBean and
                                // ContainerManagedTransaction)
implements   ProductQueryService // Business interface
{
    @Nontransactional // Or use Deployment Descriptor
    public double getPrice( String _productId )
    {
        Product.Entity productEntity = null;

        // Convenience class implements getManagerFactory()
        // within the class itself – no need to query the context.
        EntityManager entityManager =
            getManagerFactory().getEntityManager();

        // Entities are always local.
        productEntity = ( Product.Entity )
            entityManager.findEntity(
                Product.Entity.class,
                _productId
            );

        // This is also a local call.
        return productEntity.getPrice();
    }
}
```

Alternate syntax: Since this example doesn't use the `getPrimaryKey()` method, the entity reference returned by the `EntityManager` could have been cast to the business interface "Product" itself.

Also, if a separate interface file had been used instead of the inner static interface, all references to "Product.Entity" would have been replaced by "ProductEntity".

Step 2-3 – Telling the Entity Manager where to Look up the Entity

Entity references are always local. Still, it is good practice to place the lookup information in an XML file for the Entity Manager to read. This will help the Entity Manager look up the entity.

Note: This file doesn't say where an Entity **is**. It says where it can be *looked up from*. Even though the Entity is local, the lookup port needs to be specified. This is where the JNDI directory server is listening.

```
<!--“ejb-client-lookup.xml”-->
<lookups>
  <lookup>
    <class-name>myApp.*</class-name>
    <lookup-host>localhost</lookup-host>
    <lookup-port>1099</lookup-port>
  </lookup>
  ...
</lookups>
```

Sample 3 – Creating a Standalone J2SE Client (Remote client to the Service)

Step 3-1 – Defining the client

```
/* Client.java */

package clientApp;

import myApp.ProductQueryService;
import xyz.client.ManagerFactory; // Note: this is a different ManagerFactory
import xyz.ServiceManager;

public class Client
{
    public static void main( String[] _args )
    {
        String productId = _args[ 0 ];
        ProductQueryService productQueryService = null;

        ServiceManager serviceManager =
            ManagerFactory.getInstance().getServiceManager();

        // The service could be remote. We don't care. The service
        // manager hides all that.
        productQueryService = ( ProductQueryService )
            serviceManager.getService(ProductQueryService.class);

        // Again, the service could be remote,
        // and the stub implementation hides that.
        System.out.println( "Price = "
            + productQueryService.getPrice( productId ) );
    }
}
```

Step 3-2 – Telling the Client-side Service Manager where to Look up the Service

```
<!-- “ejb-client-lookup.xml” -->

<lookups>
  <!-- General rule to look up all classes matching a wildcard -->
  <lookup>
    <class-name>myApp.*</class-name>
    <lookup-host>192.168.1.18</lookup-host>
    <lookup-port>1099</lookup-port>
  </lookup>
  <!-- More specific names override more general ones -->
  <lookup>
    <class-name>myApp.client.*</class-name>
    <lookup-host>localhost</lookup-host>
    <lookup-port>1099</lookup-port>
  </lookup>
  <lookup>
    <class-name>com.somecompany.*</class-name>
    <lookup-host>10.15.236.4</lookup-host>
    <lookup-port>8080</lookup-port>
  </lookup>
  ...
</lookups>
```

The Service Manager knows from this lookup file whether to make a local or a remote call to a container to get the service stub. Accordingly, the container sends the Service Manager back a stub implementation that also performs local or remote communication back to its skeleton when it executes method calls. The application is shielded from the knowledge of whether the Service is local or remote at all times.

Sample 4 – Creating a Conversation (Local client to the Entity)

Step 4-1 – Declaring the Business Interface

```
/* ShoppingConversation.java */
package myApp;

import xyz.Conversation;

public interface ShoppingConversation
extends Conversation // Component interface for all Conversations
{
    public void addToCart( String _productId );
    public double calculateTotal();
}
```

Step 4-2 – Implementing the Business Interface

```
/* ShoppingConversationBean.java */
package myApp;

import xyz.CMTConversationBean;
import xyz.ManagerFactory;
import xyz.EntityManager;
import xyz.ConversationLifecycle;

import java.util.Collection;
import java.util.Vector;
import java.util.Iterator;

public class ShoppingConversationBean
extends      CMTConversationBean      // Convenience class
                                                // (or extend your own superclass
                                                // and implement
                                                // ConversationBean and
                                                // ContainerManagedTransaction)
implements  ShoppingConversation,      // Business interface
            ConversationLifecycle      // Optional lifecycle interface
{
    private Collection shoppingCart = null;

    // Accessor methods for all instance variables so
    // container can preserve the bean's state in session scope.
    public Collection getShoppingCart() { return shoppingCart; }
    public void setShoppingCart( Collection _shoppingCart )
    {
        shoppingCart = _shoppingCart;
    }

    // Lifecycle interface methods:
    // (Far fewer empty methods than EJB 2.x)
    public void afterInitiate()
    {
        // Needs to be done as soon as the bean is created,
        // and will last for the duration of the session.
        shoppingCart = new Vector();
    }
    public void beforeSuspend() {}
    public void afterResume() {}
    public void beforeEnd() {}

    // Business methods:
    @Nontransactional // Or use Deployment Descriptor
    public void addToCart( String _productId )
    {
```

```

// Convenience class gives you getManagerFactory()
// directly – no need to access the Context.
Product.Entity productEntity = ( Product.Entity )
    getManagerFactory()
        .getEntityManager().findEntity(
            Product.Entity.class,
            _productId );

shoppingCart.add( productEntity );
}

@Transactional // Or use Deployment Descriptor
public double calculateTotal()
{
    double total = 0.0;

    Iterator iterator = shoppingCart.iterator();
    while ( iterator.hasNext() )
    {
        Product product = ( Product ) iterator.next();
        total += product.getPrice();
    }

    return total;
}
}

```

Sample 5 – Creating a Web Tier Client (Remote Client to the Conversation)

Here are snippets from a web client component that calls the Stateful Session Bean (Conversation). The web client can either hold onto the Conversation object in the web container's HttpSession object, or it can extract the handle from the Conversation and pass it back to the client (browser), remaining stateless itself. In the latter case, when the browser sends back a fresh request with the handle as one of the parameters, the web component must recover the Conversation object using the handle and continue.

Case 5-1 – The web component holds the Conversation object in session scope

```
...  
session.putAttribute( "shoppingConversation", shoppingConversation );  
...
```

and as part of the next request,

```
...  
ShoppingConversation shoppingConversation = ( ShoppingConversation )  
    session.getAttribute( "shoppingConversation" );  
shoppingConversation.addToCart( anotherProductId );  
...
```

Case 5-2 – The web component maintains no state. The Conversation's handle gets sent all the way back to the browser

(bean component snippet)

```
...
request.setAttribute( "convHandle", shoppingConversation.getHandle() );
...
```

(JSP snippet)

```
...
<input type="hidden" name="handleToResume"
value="<%= ( String ) request.getAttribute( "convHandle" ) %>">
...
```

When the next request arrives, the bean component retrieves the handle from the HTTP request and uses it to recover the conversation

```
...
String handle = request.getParameter( "handleToResume" );
ShoppingConversation shoppingConversation = ( ShoppingConversation )
    conversationManager.findConversation(
        ShoppingConversation.class, handle );
shoppingConversation.addToCart( anotherProductId );
...
```

Statelessness in the web tier is desirable for many reasons:

1. Because state is maintained end-to-end, between front-end client and back-end business logic, any number of different channels can be used to connect them without having to code state management into each one. The application as a whole is channel-neutral, and therefore multi-channel capable.
2. A stateless web tier can be easily scaled through a web farm. No server stickiness issues arise. There is no need for expensive (in performance terms) session synchronisation or web-tier clustering.
3. The failure of a web server affects only in-flight requests being handled by that server, not all sessions maintained by that server.

The web tier and application tier could be co-located on the same physical server for performance reasons, but these architectural principles are still valid.

Sample 6 – Creating a true “Message-Driven Bean”

Step 6-1 – Declaring the Business Interface of a Service that should be invoked on receipt of a Message

```
/* ProductManagementService.java */
package myApp;

import xyz.Service;

public interface ProductManagementService
extends Service
{
    public void updatePrice( String _productId, double _price );
}
```

Step 6-2 – Implementing the Business Interface

```
/* ProductManagementServiceBean.java */
package myApp;

import xyz.CMTServiceBean;
import xyz.ManagerFactory;
import xyz.EntityManager;

public class ProductManagementServiceBean
extends      CMTServiceBean          // Convenience class
                                                // (or extend your own superclass
                                                // and implement ServiceBean,
                                                // ContainerManagedTransaction)
implements   ProductManagementService // Business Interface
{
    @Transactional // Or use Deployment Descriptor
    public void updatePrice( String _productId, double _price )
    {
        Product.Entity productEntity = null;

        // Convenience class gives you getManagerFactory()
        // directly – no need to access the Context.
        EntityManager entityManager =
            getManagerFactory ().getEntityManager();

        productEntity = ( Product.Entity )
            entityManager.findEntity(
                Product.Entity.class,
                _productId );

        productEntity.setPrice( _price );
    }
}
```

Step 6-3 – Defining the *MessageListener* class

There is no need to define an interface because it is a well-known Event Interface (“*MessageListener*”).

```
/* PriceUpdateListener.java */
package myApp;

import xyz.EventListener;
import xyz.ManagerFactory;
import xyz.ServiceManager;
import javax.jms.MessageListener;
import javax.jms.Message;

public class PriceUpdateListener
extends      SimpleEventListenerBean // Convenience class
                                                // (or extend your own superclass
                                                // and implement EventListener
                                                // and EventListenerBean)
implements  MessageListener           // Event Interface
{
    @Transactional // Or use Deployment Descriptor
    public void onMessage( Message _message )
    {
        String productId = _message.getStringProperty( “productId” );
        double price = _message.getDoubleProperty( “price” );

        // Convenience class gives you getManagerFactory()
        // directly – no need to access the Context.
        ServiceManager serviceManager =
            getManagerFactory().getServiceManager();

        ProductManagementService productManagementService =
            ( ProductManagementService ) serviceManager
                .getService( ProductManagementService.class );

        productManagementService.updatePrice( productId, price );
    }
}
```

Step 6-4 – Mapping the Listener to a Message Queue

This is done through deployment descriptors as usual.

Sample 7 – Creating a “Timer-Driven Bean” (A future-dated payment by a banking customer)

Step 7-1 – Declaring the Business Interface of the Service that should be invoked on Occurrence of the Event

```
/* PaymentService.java */
package bankingApp;

import xyz.Service;

public interface PaymentService
extends      Service
{
    public void makeImmediatePayment( String _fromAccountNumber,
                                      String _toUtilityReference,
                                      double _amount );
}
```

Step 7-2 – Implementing the Business Interface of the Service that should be invoked on Occurrence of the Event

```
/* PaymentServiceBean.java */
package bankingApp;

import xyz.CMTServiceBean;

public class PaymentServiceBean
extends      CMTServiceBean    // Convenience class
                                     // (or extend your own superclass
                                     // and implement ServiceBean
                                     // and ContainerManagedTransaction)
implements   PaymentService    // Business Interface
{
    public void makeImmediatePayment( String _fromAccountNumber,
                                     String _toUtilityReference,
                                     double _amount )
    {
        // Make a call to some legacy system...
    }
}
```

Step 7-3 – Defining the *TimeoutListener* class

There is no need to define an interface because it is a well-known Event Interface (“TimeoutListener”).

```
/* PaymentTimeoutListener.java */
package bankingApp;

import xyz.TimeoutListener;
import xyz.TimeoutEvent;

public class PaymentTimeoutListener
extends      SimpleEventListenerBean    // Convenience class
                                                // (or extend your own superclass
                                                // and implement EventListener
                                                // and EventListenerBean)
implements  TimeoutListener            // Event Interface
{
    @Transactional // Or use Deployment Descriptor
    public void onTimeout( TimeoutEvent _timeoutEvent )
    {
        // Grab all required context from the TimeoutEvent object.
        String fromAccountNumber =
            _timeoutEvent.getStringProperty(
                “fromAccountNumber” );
        String toUtilityReference =
            _timeoutEvent.getStringProperty(
                “toUtilityReference” );
        double amount =
            _timeoutEvent.getDoubleProperty( “amount” );

        // Invoke the payment service with these parameters.
        // Convenience class gives you getManagerFactory()
        // directly – no need to access the Context.
        ServiceManager serviceManager =
            getManagerFactory ().getServiceManager();

        PaymentService paymentService = ( PaymentService )
            serviceManager.getService( PaymentService.class );
    }
}
```

```
// The payment service thinks this is a normal payment
// initiated "now".
paymentService.makeImmediatePayment(
    fromAccountNumber,
    toUtilityReference,
    amount );
```

```
}
}
```

Step 7-4 – Declaring the Business Interface of the Service that sets the Timer

```
/* CustomerSelfService.java */
package bankingApp;

import xyz.Service;

public interface CustomerSelfService
extends Service
{
    public void setupPayment( java.util.Date _paymentDate,
                             String _fromAccountNumber,
                             String _toUtilityReference,
                             double _amount );
}
```

Step 7-5 – Implementing the Business Interface of the Service that sets the Timer

```
/* CustomerSelfServiceBean.java */
package bankingApp;

import xyz.CMTServiceBean;

public class CustomerSelfServiceBean
extends      CMTServiceBean    // Convenience class
                                // (or extend your own superclass
                                // and implement ServiceBean
                                // and ContainerManagedTransaction)
implements   CustomerSelfService
{
    @Transactional // Or use Deployment Descriptor
    public void setupPayment( java.util.Date _paymentDate,
                              String _fromAccountNumber,
                              String _toUtilityReference, double _amount )
    {
        // Convenience class gives you getManagerFactory()
        // directly – no need to access the Context.
        TimerManager timerManager =
            getManagerFactory().getTimerManager();
        Timer timer = timerManager.createTimer();

        // Set the timer to go off on the payment date.
        timer.setTimeout( _paymentDate );

        // Tell the timer whom to call when the timeout occurs.
        timer.addTimeoutListener( PaymentTimeoutListener.class );

        // Some information has to accompany the timeout, inside
        // the TimeoutEvent object.
        TimeoutEvent timeoutEvent = new TimeoutEvent();
        timeoutEvent.setStringProperty( "fromAccountNumber",
                                       _fromAccountNumber );
        timeoutEvent.setStringProperty( "toUtilityReference",
                                       _toUtilityReference );
        timeoutEvent.setDoubleProperty( "amount", _amount );

        // Set the TimeoutEvent of the Timer. This will be given
        // to the TimeoutListener.
        timer.setTimeoutEvent( timeoutEvent );
    }
}
```

Sample 8 – Creating Bulk Operations on a set of Entities

Operations on a set of beans (Entities) must **not** be defined within an Entity Bean. They are at a level higher than any single bean, and so they belong inside an Entity Manager. But the standard EntityManager interface only defines generic methods, so they need to be defined in a *custom* entity manager that is either a subinterface of EntityManager or an abstract class that implements it.

Step 8-1 – Defining a Custom Entity Manager Interface or Abstract Class

```
/* MyEntityManager.java */
package myApp;

import xyz.EntityManager;
import java.util.Collection;

public abstract class MyEntityManager
implements      EntityManager
{
    // This method will be implemented separately in EJB-QL:
    public abstract Collection findByPriceRange( double _fromPrice,
                                                double _toPrice );

    // This method is implemented right here in Java:
    public String[] getNamesByPriceRange( double _fromPrice,
                                          double _toPrice )
    {
        // Call the EJB-QL method:
        Collection entities = findByPriceRange( _fromPrice, _toPrice );

        String[] names = new String[ entities.size() ];
        Iterator iterator = entities.iterator();
        for ( int i = 0; iterator.hasNext(); i++ )
        {
            MyEntity myEntity = ( MyEntity ) iterator.next();
            names[ i ] = myEntity.getName();
        }

        return names;
    }
}
```

If all methods are implemented in EJB-QL, the developer only needs to define an interface and code the logic in EJB-QL within an XML file. If some methods are in Java (as in this example), then this becomes an abstract class.

Step 8-2 – Implementing the corresponding EJB-QL in an XML file

```
<!-- File "ejb-canned-queries.xml" -->
<canned-queries>
  <query>
    <class-name>MyEntityManager</class-name>

    <method-signature>
      java.util.Collection findByPriceRange( double, double )
    </method-signature>

    <query-string>
      select Object (p) from Product p
      where p.price >= ?1 and p.price <= ?2
    </query-string>
  </query>
</canned-queries>
```

Step 8-3 – Declaring the Business Interface for a Service Bean Client of the Bulk Operation

```
/* BulkOperationService.java */
package myApp;

import xyz.Service;

public interface BulkOperationService
extends Service // Identity Interface
{
    public String[] getProductsByPriceRange( double _fromPrice,
double _toPrice );
}
```

Step 8-4 – Implementing the Business Interface for a Service Bean Client of the Bulk Operation

```
/* BulkOperationServiceBean.java */
package myApp;

import xyz.CMTServiceBean;

public class BulkOperationServiceBean
extends      CMTServiceBean    // Convenience class
                                // (or extend your own superclass
                                // and implement ServiceBean
                                // and ContainerManagedTransaction)
implements BulkOperationService // Business Interface
{
    public String[] getProductsByPriceRange( double _fromPrice,
        double _toPrice )
    {
        // Tell the EntityManager which class implements the
        // method, but deal with the interface alone.
        // Convenience class gives you getManagerFactory()
        // directly – no need to access the Context.
        MyEntityManager myEntityManager = getManagerFactory()
            .getEntityManager( MyAbstractEntityManager.class );

        return myEntityManager.getNamesByPriceRange( _fromPrice,
            _toPrice );
    }
}
```

Sample 9 – Implementing Queries on a Single Entity

Step 9-1 – Using a QueryTemplate Extracted from an Entity or its Business Interface Definition

For simple queries relating to a single entity, we use a QueryTemplate object extracted from the business interface of the Entity.

```
// Get the EntityManager.
EntityManager em = context.getManagerFactory().getEntityManager();

// Get the query template corresponding to the entity or its
// Business Interface definition. In the first case, the QueryTemplate will be
// prepopulated with values extracted from the Entity.
// In the second case, it will be empty. Here, we use an empty template.
MyTemplate myTemplate = (MyTemplate) em.getQueryTemplate( MyEntity.class );

// Populate the query template with selected equality conditions (AND-ed together).
// Other properties should be ignored when the query is executed. The QueryTemplate
// and EntityManager understand which properties should be ignored, but EJB-QL
// needs to support a corresponding syntax.
myTemplate.setXXX( "value 1" );
myTemplate.setYYY( "value 2" );

// Some non-equality conditions may also be added:
myTemplate.addCondition( "someProperty",
                        "QueryTemplate.LESS_THAN",
                        "50.0" );

// The entity manager executes the query and returns a Collection of matching
// entities.
Collection entities = em.findEntities( myTemplate );

/*
// If no custom query template class (MyTemplate) exists, cast the object to the
// business interface or to the identity interface when using the respective methods.
MyData myData = (MyData) em.getQueryTemplate( MyEntity.class );
...
((QueryTemplate)myData).addCondition(...);
...
Collection entities = em.findEntities( (QueryTemplate) myData );
*/
```

Sample 10 – Distinguishing Original Objects from Copies

Step 10-1 – Declaring and Implementing the Business Interface of a Service that returns an Entity

This is the business interface of the Service. Notice that the method is declared to return the Business Interface of the Entity, not the Entity Interface.

```
public interface ProductService extends Service
{
    // Declares the return type as the Business Interface,
    // not the Entity Interface
    public Product getProduct( String _productId );
}
```

This is the implementation of the Service's business interface – the Service Bean itself. Since the method is declared to return the Business Interface, it casts the Entity reference to its Business Interface before returning it.

```
public class ProductServiceBean
implements    ProductService,    // for the business method getProduct()
             ServiceBean,        // for setServiceContext()
             ContainerManagedTransaction    // to receive a minimal
                                             //transaction controller
{
    @Nontransactional // Or use Deployment Descriptor
    public Product getProduct( String _productId )
    {
        ...
        // What the finder returns is an Entity interface.
        // The implementing class implements both the
        // Entity Interface and the Business Interface.
        // This method casts it back to the Business Interface.
        Product product = ( Product )
            entityManager.findEntity( Product.class, _productId );
        ...
        return product;
    }
}
```

Step 10-2 – Implementing a Local Client to the Service that Receives a Reference to the Entity itself

This is a local client of the Product Service (say, another service within the container's VM):

```
public Class AnotherServiceBean
implements  AnotherService,          // for the business method someMethod()
            ServiceBean,             // for setServiceContext()
            ContainerManagedTransaction // to receive a minimal
                                       // transaction controller
{
    public void someMethod()
    {
        ...
        // Acquire a reference to the Product Service
        ServiceManager sm =
            getContext().getManagerFactory().getServiceManager();
        ProductService productService =
            (ProductService) sm.getService( ProductService.class );

        // Call the ProductService to get a reference to the product.
        Product product = productService.getProduct( productId );

        // Is this a reference to an Entity or a DTO?
        assert ( product instanceof Entity ); // true

        // Treat this as an Entity.
        // All updates made to the entity are automatically persisted.
        ProductEntity productEntity = ( ProductEntity ) product;
    }
}
```

This Service receives a reference to the Entity itself. The container does not create a copy for local clients, and so does not make the transferred object implement the DTO interface.

Step 10-3 – Implementing a Remote Client to the Service that Receives a Reference to a DTO (Copy of the Entity)

This is a remote client of the product service (another tier outside the container's VM):

```
public class AnotherClient
{
    public static void main( String[] _args )
    {
        ...
        // Acquire a reference to the Product Service
        ServiceManager = ManagerFactory.getInstance()
            .getServiceManager();
        ProductService productService =
            (ProductService) sm.getService( ProductService.class );

        // Call the ProductService to get a reference to the product.
        Product product = productService.getProduct( productId );

        // Is this a reference to an entity or a DTO?
        assert ( product instanceof DTO ); // true

        // Treat this as a DTO. Updates made to it are purely local.
        ProductDTO productDTO = ( ProductDTO ) product;
    }
}
```

This client receives a reference to an object that implements the DTO interface, because the container creates copies of entities for remote clients. The client now knows that it does not hold a reference to the Entity itself.

Step 10-4 – Implementing an Application-Defined Data Transfer Object

The important thing to remember is that even if the developer is writing an object that is to be used as a Data Transfer Object by the application, it must not implement `xyz.DTO`! This interface is reserved for the framework's use, because it is meant to identify instances where the framework creates a copy of an object unknown to the client, when the client happens to be remote.

The client doesn't know that the call is remote, because the factories, managers and stubs all hide this for convenience. But the client must still know if they hold a reference to the original object or to a copy. The DTO interface makes it explicit when a copy has been created.

Developer-created DTOs must follow the interface-implementation discipline, though.

This is the Business Interface of the application's “DTO”:

```
public interface ApplicationForm
{
    public void setName( String _name );
    public String getName();
    public void setAddress( String _address );
    public String getAddress();
    ...
}
```

This is the implementation:

```
public class ApplicationFormImpl
implements ApplicationForm
{
    private String name;
    private String address;
    ...

    public void setName( String _name ) { name = _name; }
    public String getName() { return name; }
    public void setAddress( String _address ) { address = _address; }
    public String getAddress() { return address; }
    ...
}
```

If this object is ever passed to another component (the “client”), the object received by the client will implement the business interface as well as `xyz.DTO`, if the framework has made a copy of it. It will implement only the business interface and not the `xyz.DTO` interface if the client has received a reference to the original object itself. So all that the client needs to do is check if the object is an instance of the DTO interface.

Sample 11 – Implementing Persistence Interfaces

Case 11-1 – Implementing Basic Container-Managed Persistence

This is the easiest case.

```
public class MyEntityBean
extends      CMPEntityBean      // Convenience class
                                     // (or extend your own superclass
                                     // and implement EntityBean
                                     // and ContainerManagedPersistence)

implements   MyEntity           // Entity Interface
{
    // Nothing special to do
}
```

Case 11-2 – Implementing Container-Managed Persistence with Creation-time Callbacks

```
public class MyEntityBean
extends      CMPEntityBean      // Convenience class
                                     // (or extend your own superclass
                                     // and implement EntityBean,
                                     // ContainerManagedPersistence)

implements   MyEntity,         // Entity Interface
             EntityCreationLifecycle // Optional Lifecycle Interface
{
    ...
    public void beforeCreate()
    {
        // Do something, e.g., set the primary key before row insertion.
    }

    public void afterCreate()
    {
        // Do something, e.g., set up relationships (CMR).
    }
    ...
}
```

Case 11-3 – Implementing Container-Managed Persistence with all Entity Lifecycle Callbacks

If the bean is persisted by the container, but would still like to receive lifecycle callbacks anyway, this is one way to do it.

```
public class MyEntityBean
extends      CMPEntityBean    // Convenience class
                                     // (or extend your own superclass
                                     // and implement EntityBean
                                     // and ContainerManagedPersistence)
implements   MyEntity,        // Entity Interface
             EntityLifecycle  // Optional Lifecycle Interface
{
    ...
    public void beforeCreate() { /* Do something */ }
    public void afterCreate() { /* Do something */ }
    public void beforeSave() { /* Do something */ }
    public void afterLoad() { /* Do something */ }
    public void inactivityTimeoutOccurred() { /* Do something */ }
    public void inactivityPeriodEnded() { /* Do something */ }
    ...
}
```

Case 11-4 – Implementing Bean-Managed Persistence (with all Entity Lifecycle Callbacks)

Not all Entity Beans are persisted to relational databases. Some of them wrap legacy systems like mainframe-based “systems of record”. There is no alternative to Bean-Managed Persistence in such cases. The container helpfully provides callbacks based on transaction boundaries, and the bean then needs to do the appropriate read or write from the legacy store, either by itself or through a Data Access Object (DAO).

```
public class MyEntityBean
extends      SimpleEntityBean          // Convenience class
                                                // (or extend your own superclass
                                                // and implement EntityBean)
implements  MyEntity,                  // Entity Interface
            BeanManagedPersistence    // Optional Container-facing
                                                // Interface
{
    ...
    // All methods of EntityLifeCycle have to be implemented,
    // because BeanManagedPersistence is an empty
    // subinterface of EntityLifeCycle.
    public void beforeCreate() { /* Do something */ }
    public void afterCreate() { /* Do something */ }
    public void beforeSave() { /* Do something */ }
    public void afterLoad() { /* Do something */ }
    public void inactivityTimeoutOccurred() { /* Do something */ }
    public void inactivityPeriodEnded() { /* Do something */ }
    ...
}
```

Sample 12 – Implementing Transaction Interfaces

Case 12-1 – Implementing Container-Managed Transactions in a Service Bean

The bean with container-managed transaction has very limited ability to control the transaction. It can tell the transaction controller to roll back a transaction when the method returns, and it can check whether a class before this in the current thread has asked the controller for a rollback.

```
public class MyServiceBean
extends      CMTServiceBean    // Convenience class
                                     // (or extend your own superclass
                                     // and implement ServiceBean
                                     // and ContainerManagedTransaction)
implements  MyService         // Business Interface
{
    ...
    public void doSomething()
    {
        // Get hold of the transaction controller.
        // This is a very basic one with minimal control over the
        // transaction because the transaction is container-managed.
        TransactionController transactionController =
            serviceContext.getTransactionController();
        ...
        if ( transactionController.willNotCommit() )
            ...
        ...
        transactionController.dontCommit();
        ...
    }
    ...
}
```

Case 12-2 – Implementing Bean-Managed Transactions in a Service Bean

```
public class MyServiceBean
extends      BMTServiceBean    // Convenience class
                                     // (or extend your own superclass
                                     // and implement ServiceBean
                                     // and BeanManagedTransaction)
implements   MyService         // Business Interface
{
    ...
    public void doSomething()
    {
        // Get hold of the transaction controller.
        // This provides a much larger number of possible operations.
        FineGrainedTransactionController transactionController =
            ( FineGrainedTransactionController )
            serviceContext.getTransactionController();
        ...
        transactionController.begin(); // Start your own transaction

        int status = transactionController.getStatus();
        ...
        ...
        if ( /* something goes wrong */ )
            transactionController.rollback(); // Immediate rollback
        else
            transactionController.commit();
    }
    ...
}
```

Case 12-3 – Implementing Container Transaction Lifecycle in a CMT Conversation Bean

```
public class MyConversationBean
implements    MyConversation,    // Business Interface
              ConversationBean,  // Container-facing Mandatory Interface
              ConversationTransactionLifecycle // Optional Lifecycle
              // Interface
{
    ...
    // Similar to SessionSynchronization:
    public void afterTransactionBegin() { /* Do something */ }
    public void beforeTransactionEnd() { /* Do something */ }
    public boolean afterTransactionEnd() { /* Do something */ }
    ...
}
```

Sample 13 – Implementing General Bean Lifecycle Interfaces

Case 13-1 – Implementing ResourceLifecycle (For Service Beans and Event Listeners)

These are stateless components that are placed in a pool even before client requests start coming in. The container may decide to shrink the size of the pool if the volume of client requests falls over a period, so they may be withdrawn at any time.

```
public class MyServiceBean
implements    MyService,           // Business Interface
              ServiceBean,        // Mandatory Container-side Interface
              ResourceLifecycle    // Optional Interface
{
    ...
    public void afterAllocate() { /* Do something */ }
    public void beforeWithdraw() { /* Do something */ }
    ...
}

public class MyEventListener
implements    TimeoutListener,     // Event Interface
              EventListenerBean,   // Mandatory Container-side Interface
              ResourceLifecycle    // Optional Interface
{
    ...
    public void afterAllocate() { /* Do something */ }
    public void beforeWithdraw() { /* Do something */ }
    ...
}
```

Case 13-2 – Implementing ConversationLifecycle (For Conversation Beans)

```
public class MyConversationBean
implements   MyConversation,           // Business Interface
             ConversationBean,        // Mandatory Container-side
                                                // Interface
             ConversationLifecycle     // Optional Interface
{
    ...
    public void afterInitiate() { /* Do something */ }
    public void beforeSuspend() { /* Do something */ }
    public void afterResume() { /* Do something */ }
    public void beforeEnd() { /* Do something */ }
    ...
}
```

About the Authors

Rajat Taneja is an architect with Zurich Financial Services, Sydney, Australia. He has many years of J2EE experience and has developed EJB-based applications in very large and complex environments. Many of the core ideas in this model stem from his practical experience with EJBs in demanding contexts. His intolerance for sloppy and impractical design has brought a level of discipline to the model and kept it lean and free of crud. Rajat is a Sun Certified Java Programmer and a Sun Certified Enterprise Architect. Rajat can be reached at rajattaneja@optusnet.com.au.

Ganesh Prasad is an architect with Westpac Banking Corporation, Sydney, Australia. He has many years of experience with J2EE in large banking applications. Many of the core ideas in this model came, however, when he was studying for the Sun Certified Business Component Developer exam, when he had the opportunity to study the current EJB component model in great detail, warts and all. Ganesh is a Sun Certified Java Programmer and a Sun Certified Web Component Developer, but will not endorse the EJB 2.x design now by taking the Business Component Developer exam till the component model is sorted out! He, however, recommends the excellent book “Head First EJB” by Kathy Sierra and Bert Bates to anyone wishing to understand how EJBs work today. Ganesh can be reached at sashi@easy.com.au.

About the Reviewers

Vikrant Todankar is a senior consultant with EDS Australia, on assignment to the Commonwealth Bank of Australia, Australia's second-largest bank. He has several years of experience building J2EE applications in the Financial Services sector. Vikrant is a “code rat”, who prefers reading source code to documentation. Ask Vikrant how Tomcat, JBoss or Hibernate implements some feature, and he will burrow into the innards of the code and ferret out their dirtiest implementation hacks. Vikrant provided some much-needed reality checks on this model.

Simon Knudsen is Lead Designer, Java Centre of Excellence, Westpac Banking Corporation, Sydney, Australia. He writes some of the cleanest code in the business, and is a standards champion and best practice fanatic. Having his stamp of approval on this model means a lot.

Appendix A – A Laundry List of Litanies (EJB 2.x)

Let us quickly recapitulate the four fundamental flaws of the EJB 2.x model:

- Flaw 1. It forces Entity Beans, Stateless Session Beans and Stateful Session Beans to implement common interfaces when they are in fact very different animals. These interfaces are simultaneously anaemic and overloaded, forcing implementations to either repeatedly provide the same functionality, or implement irrelevant functionality.
- Flaw 2. It fails to separate operations *on a set of beans* from operations *on a single bean*, forcing an individual bean to implement functionality relevant to itself as well as functionality affecting a set of similar beans.
- Flaw 3. It creates unnecessary dependencies between components, breaking encapsulation in a number of places.
- Flaw 4. It overloads terms with multiple meanings in different contexts, confusing developers.

These four flaws are directly or indirectly responsible for all the problems that developers face with EJBs.

1. Why do Stateless Session Beans expose a home method called “remove()” to the client? It's not the client that removes Stateless Session Beans, it's the container that removes them when it wants to shrink the pool. The anomaly exists because all types of beans have to shadow the same home interfaces EJBHome and EJBLocalHome, and “remove()” is one of the methods in those interfaces. (Flaw 1)
2. Why do Session Beans (Stateful or Stateless) expose a component method called “getPrimaryKey()” when only Entity Beans have primary keys? This is because all types of beans have to shadow the same component interfaces EJBObject and EJBLocalObject, and “getPrimaryKey()” is one of those methods. (Flaw 1)
3. Why should all kinds of beans expose a component method called “isIdentical()”? This always returns true for Stateless Session Beans and always returns false for Stateful Session Beans. It really only makes sense for Entity Beans, and even there, its value as a bean method is dubious. It belongs at a higher level than either bean (Flaws 1 and 2)
4. Why should beans “shadow” the home interfaces but not implement them? The home interfaces list methods that the developer cannot implement (e.g., getHomeHandle() and getEJBMetaData()), so the bean shouldn't implement the interface, just “shadow” it to implement the ones it can. (Follows from Flaw 1)
5. Why should beans “shadow” the component interfaces but not implement them? The component interfaces list methods that the developer cannot implement (e.g., getHandle(), getEJBHome() and getEJBLocalHome()). (Follows from Flaw 1)
6. Why should code pertaining to a set of beans reside within a single bean's home interface, and why should the bean define corresponding methods? There should be something “above” the level of a bean that handles logic affecting more than one bean. (Flaw 2)
7. Why should the lifecycle methods and bulk operations be shadowed by an individual bean? (Follows from Flaws 1 and 2)
8. Why should the lookup logic be JNDI -> Home Interface -> Component Interface? Why can't it be a simpler Manager -> Business Interface? (Follows from Flaw 2)
9. Why should a client application that is not in the container have to bundle libraries specific to the app server vendor in order to invoke an EJB? EJB is a vendor-independent

- technology, right? If EJBs are moved to a container from another vendor, all the client applications have to be rebuilt with a different set of “jar” files! (Follows from the previous point)
10. Why should client-side lifecycle methods and bulk methods share the same interface? All client-side lifecycle methods are common to a bean type (Entity, Stateless Session or Stateful Session), whereas bulk methods are necessarily unique to a particular Entity Bean. (Follows from Flaw 2)
 11. When the client-side lifecycle method “create()” is called for a Stateful Session Bean, the container creates it and calls the callback method “ejbCreate()” on it. But the callback method “ejbCreate()” is called only once in the lifetime of a Stateless Session Bean, when it is first allocated to a pool, not whenever a client requests access to one through “create()”. The methods need not be synchronised. They shouldn't even be named the same way. (Follows from Flaws 1, 3 and 4)
 12. Why should Stateless Session Beans implement “ejbActivate()” and “ejbPassivate()”? Merely because they are methods in a common interface they share with Stateful Session Beans? (Flaw 1)
 13. Why should Beans have to implement all methods in the respective callback interfaces when they're empty most of the time anyway? (Follows from Flaw 2)
 14. Why should business logic sit inside a Message-Driven Bean? What if the same logic needs to be invoked synchronously as well? (Flaw 3)
 15. Why should a Stateless Session Bean implement `TimedObject` and its method “onTimeout()”? Why should a bean that is meant to encapsulate business logic need to know about calling mechanisms? (Flaw 3)
 16. Why should all beans have the method “getMessageContext()” in their contexts? This pertains to Web Services through a messaging interface, but why should a bean know how it has been called? And why should all types of beans have this method? (Flaws 1 and 3)
 17. The “ejbCreate()” method of an Entity Bean is called at the time a row is inserted in a table. The “ejbCreate()” method of a Stateful Session Bean is called when it is being created for a client. The “ejbCreate()” method of a Stateless Session Bean is called once at the beginning of its lifecycle when the container creates a whole bunch of beans. These are totally different methods, so why do they have the same name? Similar arguments apply to “ejbRemove()”. (Flaw 4)
 18. The “ejbPassivate()” method of an Entity Bean is called after a period of inactivity, to tell the bean that it is going back into the pool. The “ejbPassivate()” method of a Stateful Session Bean is called when the bean is about to be serialised to disk. These are again totally different methods, so why do they have the same name? Similar arguments apply to “ejbActivate()”. (Flaw 4)
 19. Why is the “ejbCreate()” method of an Entity Bean declared as returning a `String`, when it must actually be implemented to return a `null`? Yes, yes, it's for backwards-compatibility with EJB 1.1, which foolishly assumed that primary keys should be strings, and that the key should be returned by the “ejbCreate()” method. But we're in EJB 3.0 land now, and the problem is still not getting fixed... (Flaw 3)
 20. How can a bean developer write a Session Bean without knowing whether it is going to be deployed as Stateful or Stateless? Is this really a Deployment Descriptor entry? (Flaw 1)
 21. Message-Driven Beans (as a description of end-to-end functionality) are not first-class EJBs. The architecture involves an Event Listener that responds to an asynchronous event (like a message or a timeout) and then calls business logic in a Stateless Session Bean. Since the same logic may need to be triggered by the arrival of a message, by a timeout, and may also need to be called synchronously, encapsulation demands that it live inside a

Stateless Session Bean. The event listener must only unmarshall parameters from the event and use them in the subsequent call to the Stateless Session Bean. That's MDB best practice. It's wrong to apply the term "Message-Driven Bean" just to the Event Listener component (Flaw 4)

22. We don't know where to begin about EJB 2.1 Timers. Timers get a 10 out of 10 for bad design. For starters, Entity Beans should not receive timeout events. Events should trigger business logic and not affect data directly. (Flaw 3)
23. It is meaningless for Message-Driven Beans to receive timeout events. What are these beans driven by again? (Flaw 3)
24. If a Stateless Session Bean sets a Timer, why does the resulting timeout attach to an instance of the same Stateless Session Bean class? There is an implied two-way dependency that isn't justified. The component that sets a timer should be independent of the component that receives the timeout event. (Flaw 3)
25. Why can't a Stateful Session Bean *set* a Timer? It gets an Exception if it as much as tries to call `getTimerService()` on its context. It may not make sense for a Stateful Session Bean to receive timeout events (or it may, if you ask gamers), but why should it be prevented from *setting* timers? (Flaw 3)