EJB™ 2.1 Kick Start

International Standard Book Number: 0-672-32178-5

Warning and Disclaimer

When reviewing corrections, always check the print number of your book. Corrections are made to printed books with each subsequent printing. To determine the printing of your book, view the copyright page. The print number is right-most number on the line below the "First Printing" line. For example, the following indicates the 4th printing of a title.

First Printing: October 2002

06 05 04 03 10 9 8 7 6 5 4

Misprint	Correction
Introduction, page 2, fourth and fifth bulleted points at the top of the page: • You should have access to the Java Development Kit (JDK), version 1.**3** or higher. The JDK is available for download at www.javasoft.com. • You need a copy of **Sun's J2EE Reference Server (available at www.javasoft.com),** and a Windows platform (NT, 2000, or XP) on which to run the server.	• You should have access to the Java Development Kit (JDK), version 1.**4** or higher. The JDK is available for download at www.javasoft.com, **and is also bundled with BEA's WebLogic Server 8.1 (and higher)**. • You need a copy of **BEA's WebLogic Server, version 8.1 or higher (available at www.bea.com**), and a Windows platform (NT, 2000, or XP) on which to run the server. • You should have a copy of the Jakarta Project's ANT utility. ANT is the predominant software-building tool in the J2EE and J2SE arenas, and both the author and WebLogic Server 8.1 rely heavily on ANT for compiling and deploying J2EE components. ANT is available at www.jakarta.org and is also bundled with WebLogic Server.
Page 10, add the following point after the third bulleted point, which begins "The ability to charge purchases..." • Bulk purchasing capabilities—Corporate and institutional customers' purchasing systems can interact autonomously with BookEaz via a b2b (business to business) mechanism. Same page, add the following after the "Same-day shipping..." bulleted point: • Automated shipment tracking—The system will accept changes in shipment status directly from major shippers.	
Page 13, Table 1.1, update to the following--bold text has been added: **Actor Name Actor Type Description**	

Shopper	Person	A customer or visitor to the BookEaz Web site.	
Corporate Customer	**Person**	**Corporate Customers purchase books in bulk. They share many traits with Shopper, so they are considered to be a "subclass" of Shopper.**	
Administrator	Person	A person who modifies or maintains the BookEaz Web site.	
Shipper	**System**	**A package delivery company through which BookEaz ships orders to its customers.**	
Order Fulfillment	System	An external system that will be purchased separately from the BookEaz system.	
Email **Notification** System	System	A conduit through which the system transmits email messages to Shoppers.	
Page 15, Paragraph just before Table 1.4: BookEaz is purchasing a third-party order fulfillment system. Because the system doesn't perform order fulfillment, it is modeled as an actor (see Table 1.4).			BookEaz is purchasing a third-party order fulfillment system. Because the system doesn't perform order fulfillment, it is modeled as an actor (see Table 1.4). **However, implementation-specific knowledge, such as "BookEaz is buying an off-the-shelf order fulfillment system," is intentionally ignored at the use-case level; use cases ignore *how* a system is implemented, focusing instead on *what* features the system needs to provide.**
Page 15, Table 1.4 Second column: **Forwarding** an Order			**Fulfilling** an Order

page 15, insert the following after Table 1.5:

Corporate Customer's Candidate Use Cases

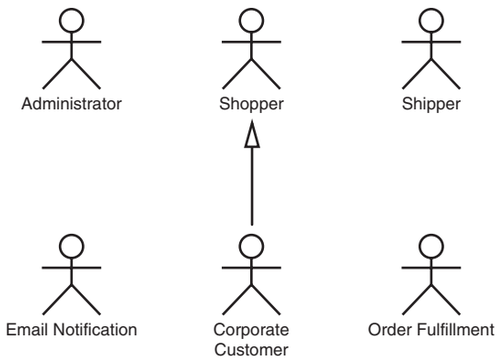
BookEaz's corporate customers need to autonomously submit orders under the auspices of a b2b (business-to-business) paradigm. Corporate customers' procurement systems can interact directly with BookEaz without any human intervention.

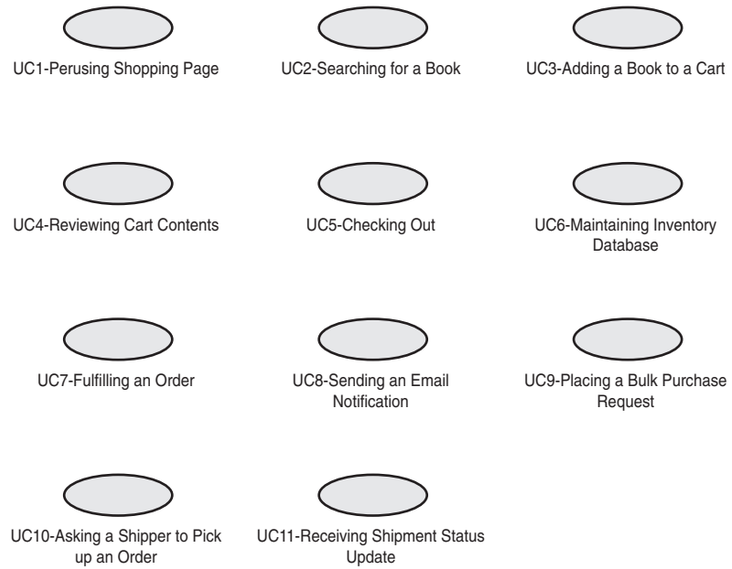
Use Case Number	Use Case Name	Description
UC9	Placing a Bulk Purchase Request	Corporate customers' procurement systems interact with BookEaz via a b2b mechanism.

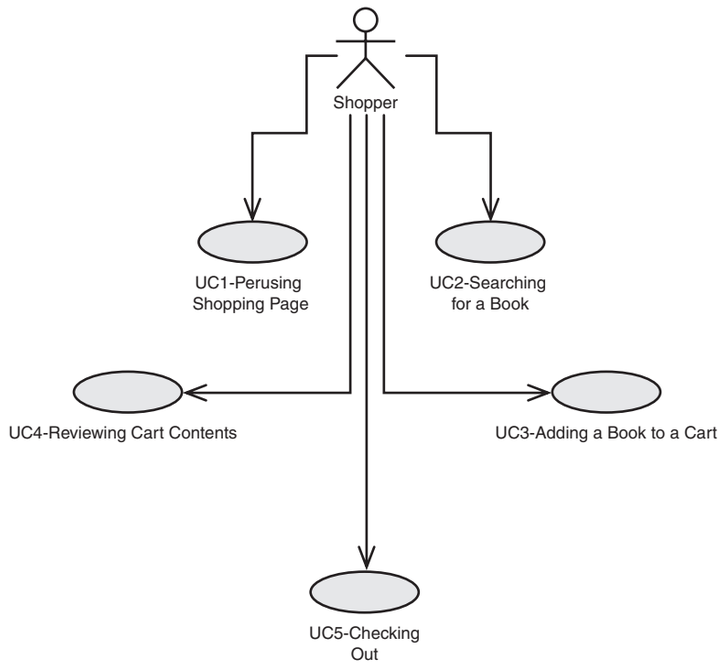
Shipper's Candidate Use Cases

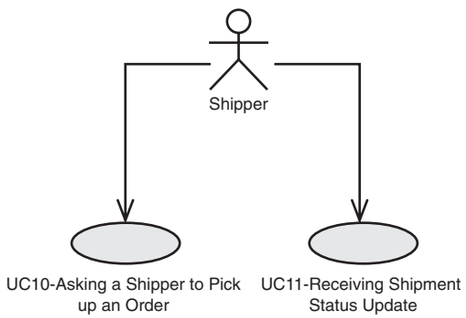
BookEaz intends to ship orders via next-day delivery companies such as UPS, FedEx, and the United States Postal Service. These shippers interact with BookEaz in two ways, as described in the following table.

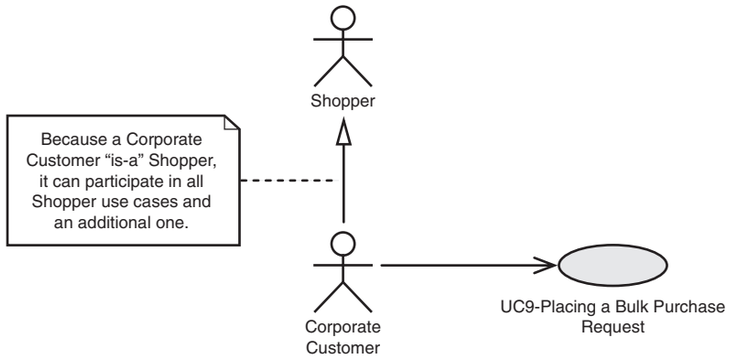
Use Case Number	Use Case Name	Description
UC10	Asking a Shipper to pick up an Order	Shippers must be notified when an order is ready to leave BookEaz's fulfillment center.
UC11	Receiving Shipment Status Update	Shippers can notify BookEaz of significant changes in a shipment's status, such as the shipment's current location.

Page 16, first line of the second paragraph: After choosing a modeling tool (I used Rational Rose **2001**)...	After choosing a modeling tool (I used Rational Rose **2002**)...
Page 16, Figure 1.3, figure and caption change: **Figure 1.3** *The BookEaz analysis model houses **four** actors, which are depicted in the actor dictionary diagram.*	 **Figure 1.3** The BookEaz analysis model houses **six** actors, which are depicted in the actor dictionary diagram.

Page 17, Figure 1.4	Replace with the following new figure: 

Page 18, Figure 1.5	Replace with the following new figure: 
Page 18, paragraph after Figure 1.6: Fulfillment is largely the responsibility of a COTS system, but BookEaz proper interacts with that third-party software, so Fulfillment is rendered as an actor. This actor's **single interaction with BookEaz is encapsulated in the Forwarding an Order use case, as shown in** Figure 1.7.	Fulfillment is largely the responsibility of a COTS system, but BookEaz proper interacts with that third-party software, so Fulfillment is rendered as an actor. This actor's **two interactions with BookEaz are shown** in Figure 1.7.

Page 19, Figure 1.8 caption: **Figure 1.8** *This diagram shows Email **System** and its use cases.*	**Figure 1.8** *This diagram shows Email **Notification** and its use cases.*
Page 19, add the following text and figures after Figure 1.8: Shippers interact with BookEaz in two ways: They are notified when shipments are ready to be picked up and sent to customers, and they provide status updates back to BookEaz as shipments move through their lifecycle.  **Figure 1.9** *Shippers collaborate with BookEaz to deliver customer orders.* Finally, Corporate customers interact with BookEaz in much the same manner as normal shoppers, but they also are capable of placing bulk purchase orders.	

 **Figure 1.10** *Corporate customers are shoppers who can also place bulk orders.*	
Page 27, heading halfway down the page: "Use Case 7: **Forwarding** an Order"	"Use Case 7: **Fulfilling** an Order"
Page 28, add the following text just before the "Conclusion" heading: **Use Case 9: Placing a Bulk Purchase Request** **Synopsis:** Corporate customers send bulk purchase requests directly from their procurement systems to BookEaz's shopping portal. Use Case 9 "includes" use case Use Case 5, which means that Use Case 9 eventually replicates the behavior of Use Case 5. This inclusion of another use case occurs near the end of Use Case 9's primary flow. **Preconditions** • The Corporate customer must be registered as such at BookEaz. • The Corporate customer must have a user name, a	

password, and a billing address on file with BookEaz.

Primary Flow

- The Corporate customer sends its user name, password, and a list of books to the system.
- The system authenticates the user name/password combination. If the user name/password combination is invalid, then go to Alternative Flow 1.
- The system creates a shopping cart for the Corporate customer.
- The system places each designated item into the cart.
- The system sends the cart to the checkout process, which is described in Use Case 5.
- The system sends a status message (either success or failure) to the Corporate customer.
- Use Case 9 ends.

Alternative Flow 1—Invalid User Name or Password

- The system sends an "invalid user name or password" error message to the Corporate customer.
- Use Case 9 ends.

Use Case 10: Asking a Shipper to Pick Up an Order

Synopsis: When an order is ready to be shipped, BookEaz's fulfillment system notifies the most appropriate shipper of that fact; as a result, the shipper picks up the order and ships it to the customer. Customers designate their preferred shipping method at checkout time; that is how the most appropriate shipper is selected.

Preconditions

- Customer designated a preferred shipping method at checkout time.
- Order is fulfilled.
- Order is ready to be shipped.

Primary Flow

- The fulfillment system notifies BookEaz's main system that an order is ready to be shipped.
- The system examines the order and finds the shipper designated thereon.
- The system determines the communications method to use for contacting the designated shipper.
- The system publishes a "shipment ready for pickup" message to the designated shipper.
- Use Case 10 ends.

Use Case 11: Receiving Shipment Status Update

Synopsis: Virtually all shipping companies track every aspect of a shipment's lifecycle, from the moment it is picked up at BookEaz until the moment it arrives at the customer's door. Shippers are able to communicate these lifecycle events to BookEaz, which in turn can communicate them to the customer. This allows customers to track their shipments at BookEaz's Web site instead of having to navigate to the shipper's site.

Preconditions

- Shipper is capable of communicating lifecycle events to BookEaz.

• Shipper has picked up shipment from BookEaz. **Primary Flow** • A significant event occurs that changes a shipment status; for example, the shipment might have been dropped off at an airport hub for shipment to the customer's city. • The shipping company publishes a message to BookEaz that details the event. • The system receives the message from the shipping company. • The system saves the status information so that it can be retrieved by the customer. • Use Case 11 ends.	
Page 32-33, last paragraph at the bottom of page 32: Because you are going to be implementing BookEaz on the twin pillars of a Web server and an EJB container, you might as well acquire one product that serves both purposes. Several products, both commercial and open-source, act as both Web servers and EJB containers. Unfortunately (or fortunately, if you work for BEA), only two products are currently available that fully adhere to the EJB **2.0** specification, Sun's J2EE Reference Server and BEA's WebLogic Enterprise Server (versions **6.0** and higher). This book uses **Sun's** server for all its examples, **but they will also work on WebLogic**. Other servers are available for downloading; please refer to **Appendixes B and C** for instructions on acquiring and installing **these products**.	Because you are going to be implementing BookEaz on the twin pillars of a Web server and an EJB container, you might as well acquire one product that serves both purposes. Several products, both commercial and open-source, act as both Web servers and EJB containers. Unfortunately (or fortunately, if you work for BEA), only two products are currently available that fully adhere to the EJB 2.1 specification, Sun's J2EE Reference Server and BEA's WebLogic Enterprise Server (versions 8.1 and higher). This book uses WebLogic's server for all its examples. Please refer to Appendix B for instructions on acquiring and installing WebLogic.
Page 34, second paragraph from the end: From your (the programmer's) perspective, an Entity Bean is composed of three classes/interfaces that collaborate under the auspices of the EJB Entity Bean design pattern: the *home interface*, the ***remote*** *interface*, and the *implementation*...	From your (the programmer's) perspective, an Entity Bean is composed of three classes/interfaces that collaborate under the auspices of the EJB Entity Bean design pattern: the *home interface*, the ***component*** *interface*, and the *implementation*...

Page 36, "Entity Bean Remote Interface" heading: **Entity Bean** Remote **Interface** The second **component** of an entity troika is the **remote** interface. The **remote** interface, from the perspective of client code, *is* the entity; it is the interface through which the entity's attributes are accessed. The `Book` entity's **remote** interface is named `Book`.	**Entity Bean** Component **Interface** The second **part** of an entity troika is the **component** interface. The **component** interface, from the perspective of client code, *is* the entity; it is the interface through which the entity's attributes are accessed. The `Book` entity's **component** interface is named `Book`.
Page 37, first paragraph: The reference to EJB 2.0 should refer to EJB 2.1.	
Page 40, just after Figure 2.7, insert the following: ***Note*** Figure 2.7 shows that a `ShoppingCart` is associated with zero or more `ShoppingItems`. The small box attached to `ShoppingCart` makes this relationship a *qualified association*. A qualified association is the object equivalent of a database foreign-key relationship. Figure 2.7's qualified association states that a `ShoppingCart` is associated with its `ShoppingItems` via the `ShoppingItems'` SKU (stock keeping unit) attributes. See Appendix A for more information about qualified associations.	
Page 42, text at the top of the page: • Instead of a **remote interface**, a Local Entity Bean has a **local interface** (this makes sense because Local Entity Beans are not remotely accessible). • A Local Entity Bean's **local** interface extends `javax.ejb.EJBLocalObject` instead of `javax.ejb.EJBObject`.	• Instead of a **remote component interface**, a local Entity Bean has a **local component interface** (this makes sense because local Entity Beans are not remotely accessible). • A local Entity Bean's **component** interface extends `javax.ejb.EJBLocalObject` instead of `javax.ejb.EJBObject`.

Page 42, add the following after Figure 2.9: ***Note*** An interface can be denoted by a stereotype label (for example, EJBLocalHome), by a decoration (such as EJBLocalObject), or by a stereotype icon (for example, EntityBean). If you don't need to show an interface's methods on a diagram, consider depicting it via its stereotype icon. The stereotype icon is relatively small, thereby helping maintain a minimally sized diagram. Refer to Appendix A for information about interfaces and stereotypes.	
page 43, paragraph at the top of the page, second line/sentence: ...As you have learned more about how ShoppingItems are used, however, it has become evident that although there is no need to make ShoppingItem's methods remotely accessible, its **data** needs to be remotely viewable.	As you have learned more about how ShoppingItems are used, however, it has become evident that although there is no need to make ShoppingItem's methods remotely accessible, its **attributes** need to be remotely viewable.
Page 43, add the following below Figure 2.10: ***Note*** Use sequence diagram narratives to denote loops and branches, as shown on Figure 2.10. Text just below the existing Figure 2.10: ShoppingItemData is a serializable class, which means that its instances can be transported across the network. ShoppingItem is modified to contain an instance of ShoppingItemData, and all of ShoppingItem's attributes are moved into ShoppingItemData. ShoppingItem is augmented with a getData() method, which returns ShoppingItem's ShoppingItemData instance. **Finally, ShoppingCart's getItems() method is modified to return a collection of ShoppingItemData objects instead of a collection of ShoppingItem**	ShoppingItemData is a serializable class, which means that its instances can be transported across the network. ShoppingItem is modified to contain an instance of ShoppingItemData, and all of ShoppingItem's attributes are moved into ShoppingItemData. ShoppingItem is augmented with a getData() method, which returns ShoppingItem's ShoppingItemData instance. **Finally, the getItemDataList() method is added to ShoppingCart; this method returns a collection containing ShoppingItemData renditions of the cart's ShoppingItems**. With these

objects. With these modifications in place, client code that wants to view ShoppingItems' data portions is able to do so, as seen in SD4.0.	modifications in place, client code that wants to view ShoppingItems' data portions is able to do so, as seen in SD4.0.
Page 46, first sentence of the paragraph at the top of the page: The Session Bean home interface provides methods for creating instances of Session Bean **remote** interfaces.	The Session Bean home interface provides methods for creating instances of Session Bean **component** interfaces.
Page 46, "Session Bean Remote Interface" heading and text: **Session Bean** Remote **Interface** The second component of a Session Bean is the **remote** interface. The **remote** interface, from the perspective of client code, *is* the Session Bean; it is the interface through which the bean's services are accessed by calling its methods. For example, To "check out" (that is, to complete the purchase process), Shoppers **invokChapter 2 correx** Session Bean **remote** interfaces tend to be named <*bean-name*>, much like Entity Bean **remote** interfaces. The Checkout Session Bean, for example, has a **remote** interface named Checkout.	**Session Bean** Component **Interface** The second component of a Session Bean is the **component** interface. The **component** interface, from the perspective of client code, *is* the Session Bean; it is the interface through which the bean's services are accessed by calling its methods. For example, To "check out" (that is, to complete the purchase process), Shoppers **invoke checkout().** Session Bean **component** interfaces tend to be named <*bean-name*>, much like Entity Bean **component** interfaces. The Checkout Session Bean, for example, has a **component** interface named Checkout.
Page 46, paragraph below the "Session Bean Implementation" heading: The final piece of the Session Bean puzzle is the implementation. The implementation class lives in the EJB container and houses the implementations of all the methods declared in the Session Bean home interface and the Session Bean **remote** interface as well as several methods the EJB specification mandates.	The final piece of the Session Bean puzzle is the implementation. The implementation class lives in the EJB container and houses the implementations of all the methods declared in the Session Bean home interface and the Session Bean **component** interface as well as several methods the EJB specification mandates.
Page 46, Figure 2.12 caption: **Figure 2.12** *Session Beans such as* ***Checkout*** *consist of a home interface, a* ***remote*** *interface, and an implementation.*	**Figure 2.12** *Session Beans such as* ***Checkout*** *consist of a home interface, a* ***component*** *interface, and an implementation.*

Page 46, add the following after Figure 2.12: ***Note*** Figure 2.12 uses the dependency relationship (a dashed line with an open arrow tip) to indicate that `ShoppingCartBean` uses `ShoppingCart`, `PaymentProcessor`, and `OrderHome`. These dependency relationships imply that `ShoppingCartBean`'s Java implementation will contain import statements for these three classes. As you discover dependency relationships, as you will on Figure 2.13, you should immediately add them to the logical model.	
Page 47, add the following after Figure 2.13: ***Note*** External classes and systems are often depicted as *boundary classes*; for example, Figure 2.13 uses the boundary stereotype to indicate that `PaymentProcessor` is an external system. Refer to Appendix A for more information about boundary classes.	
Page 49, paragraph after the "Discoveries" heading: All three instances of `BookAdmin` should refer to `BookAdministrator`.	
Page 49, last paragraph on the page, continuing to the following page: Both instances of EJB 2.0 should refer to EJB 2.1.	
Page 50, paragraph below the "Discoveries" heading: Message 1 of SD7.0 depicts an *asynchronous method invocation*, which means that Joe's `Order` doesn't wait around for a return message from **`FulfillmentManager`**; it just sends itself as a	Message 1 of SD7.0 depicts an *asynchronous method invocation*, which means that Joe's `Order` doesn't wait around for a return message from **`FulfillmentQueue`**; it just sends itself as a message, and

message, and then returns to other processing tasks.	then returns to other processing tasks.
Page 50, add the following just before the "Message-Driven Beans" heading: ***Note*** Asynchronous method invocations, such as the call to `send` in Figure 2.16, are depicted with an arrow that is missing the bottom half of its arrowhead.	
Page 51, last sentence of the paragraph just above "Java Message Service (JMS) and Message-Driven Beans" heading: ...Instead of interacting directly with Message-Driven Beans, producers/publishers indirectly access them by calling the **`publish()`** method on a class called `javax.jms.`**`TopicPublisher`**.	Instead of interacting directly with Message-Driven Beans, producers/publishers indirectly access them by calling the **`send()`** method on a class called `javax.jms.`**`QueueSender (or by calling the publish() method on a class called javax.jms.TopicPublisher).`**
Page 51, last sentence of the paragraph under "Java Message Service (JMS) and Message-Driven Beans" heading: ...`OrderBean`, for example, plays the role of publisher/producer in SD7.0, so to push a message to `FulfillmentManager`, it has to use the **`publish()`** method of the `javax.jms.`**`TopicPublisher`** class	`OrderBean`, for example, plays the role of publisher/producer in SD7.0, so to push a message to `FulfillmentManager`, it has to use the **`send()`** method of the `javax.jms.`**`QueueSender`** class.
Page 52, paragraph just above Figure 2.19: Sequence diagramming is still revealing interesting aspects of your design, even as the end of your first design iteration approaches. SD8.0 reveals a new Message-Driven Bean named `EmailManager`, which `FulfillmentManager` uses to communicate with customers. `EmailManager`'s design is depicted in Figure 2.19.	Sequence diagramming is still revealing interesting aspects of your design, even as the end of your first design iteration approaches. SD8.0 reveals a new Message-Driven Bean named `EmailManager`, which `FulfillmentManager` uses to communicate with customers. `EmailManager`'s design is depicted in Figure 2.19. **Like so many frequently utilized services, email facilities are provided as a standard part of J2EE. As you will see when you delve into `EmailManager`'s code, these off-the-shelf facilities significantly lessen the burden placed on developers**.
Page 53, first paragraph: SD8.0 uncovers the existence of two new `Order` attributes: `emailAddress` and `status`. Naturally, you add these attributes and	SD8.0 uncovers the existence of two new `Order` attributes: `emailAddress` and `status`. Naturally, you add these attributes and

their concomitant getters and setters to the `Order` Entity Bean as soon as you discover them. Adding an attribute to an Entity Bean requires adding the attribute's getter and setter to the Entity Bean's **remote** interface and its implementation. This dual-change method also holds true when adding methods to Session Beans.	their concomitant getters and setters to the `Order` Entity Bean as soon as you discover them. Adding an attribute to an Entity Bean requires adding the attribute's getter and setter to the Entity Bean's **component** interface and to its implementation. This dual-change method also holds true when adding methods to Session Beans.
Page 53, add the following text just before the "Conclusion" heading:	

SD9.0: Placing a Bulk Purchase Request

UC9 stipulates that corporate customers can place large orders potentially containing many different books. BookEaz's development staff has no guarantee that its corporate customers' computers can access J2EE services; that is, there is no guarantee that external companies use Java as their programming language. When two computer systems need to exchange information, and there is no guarantee that they both "speak the same language" (Java, in this instance), you need to employ some sort of language-neutral information interchange protocol to allow them to interact. There are many such protocols in use today, but being a modern J2EE shop, BookEaz decided to bridge this potential communication gap by using Web Services.

Web Services provide a way for distributed, potentially disparate computers to exchange information. In BookEaz's case, the information being exchanged is bulk orders for books, but virtually any sort of information can be exchanged via a Web Service.

In order for two businesses to interact via a Web Service, they both must agree to use a neutral language when sending and receiving information; only by abiding by this agreement can the two parties expect to conduct business. For example, BookEaz's systems are largely Java-based, but a corporate customer, such as Acme Corp. in Figure 2.20, might use a combination of Microsoft Excel and VBA (Visual BASIC for Apps) to produce bulk orders. BookEaz cannot expect Acme Corp. to abandon its Microsoft-centric order-creation process, nor can Acme Corp. expect BookEaz to abandon its Java-centric ways and replace them with a Microsoft solution.

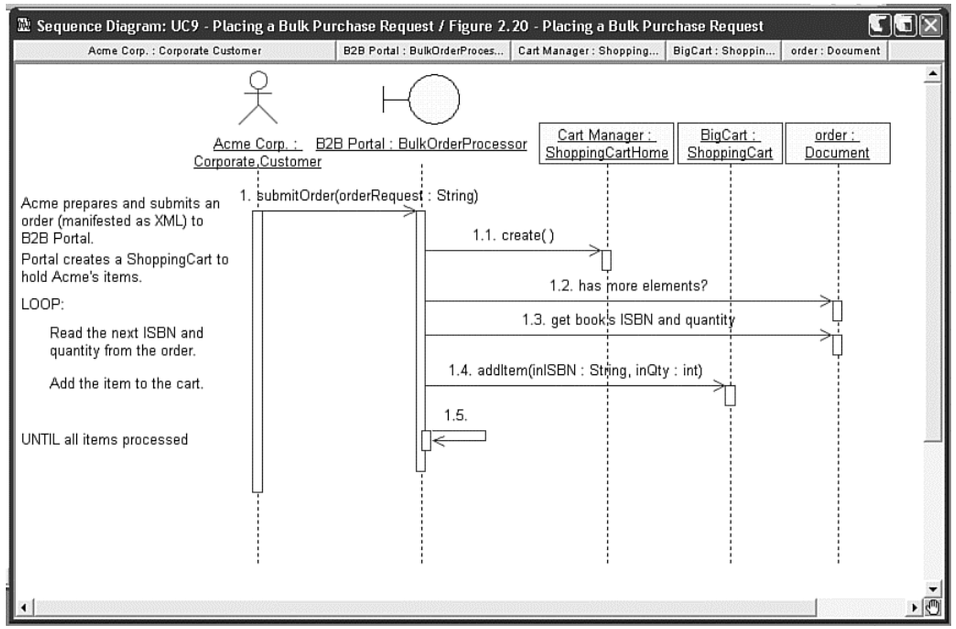


Figure 2.20

BookEaz offers b2b facilities for corporate customers.

XML is the Lingua Franca of Web Services. If a corporate customer such as Acme Corp. can transform bulk orders into XML, they can transmit those orders to BookEaz's bulk-order processor. If BookEaz uses WebLogic, they can provide a bulk-order processor Web Service that accepts orders as XML documents and pushes them into a Session Bean for further processing. Simply inserting a Web Services layer allows any XML-capable corporate customer to use BookEaz.

Discoveries

SD9.0 depicts BookEaz's first Web Service, `BulkOrderProcessor`, which provides a means for corporate customers to place bulk orders, regardless of the computing platforms used by those customers. Your first impression might be that implementing something as sophisticated as a Web Service will be an arduous task, but it is not. In fact, implementing `BulkOrderProcessor` in WebLogic is exceedingly simple; you provide a simple Session Bean or Message-Driven Bean implementation, and WebLogic generates a Web Service front end on your behalf.

Stakeholders who view SD9.0 are largely unconcerned with the specifics of `BulkOrderProcessor`'s implementation, so the fact that it is composed of a Session Bean with a Web Service front end is intentionally omitted from the sequence diagram.

This information is of paramount importance to those who have to implement `BulkOrderProcessor`, however, so it is immediately captured on a class diagram, as shown in Figure 2.21.

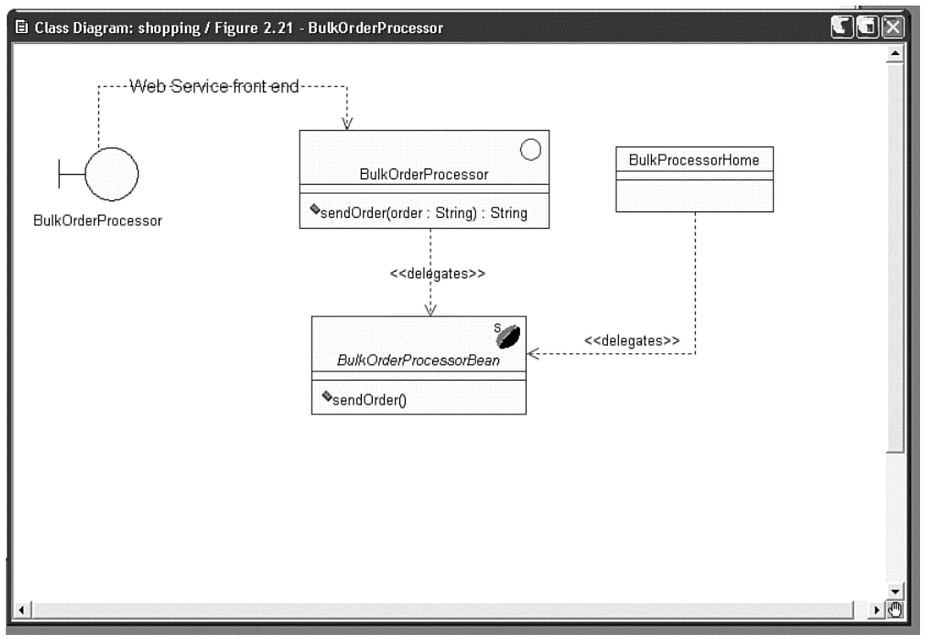


Figure 2.21

`BulkOrderProcessor` is a Web Service implemented as a Session Bean.

SD10.0: Asking a Shipper to Pick Up an Order

BookEaz employs a variety of shipping services, such as UPS and FedEx, to deliver orders to customers. The act of fulfilling an order (that is, the act of gathering books together and placing them in a shipping container) is performed by BookEaz's off-the-shelf fulfillment system. The fulfillment system lacks a mechanism for notifying shippers when orders are ready to be picked up from BookEaz's warehouse, however, so you need to develop this functionality yourself. `ShipperNotifier`, which turns out to be a Message-Driven Bean (see Figure 2.22), is the component that provides the means for the fulfillment system to interact with shippers.

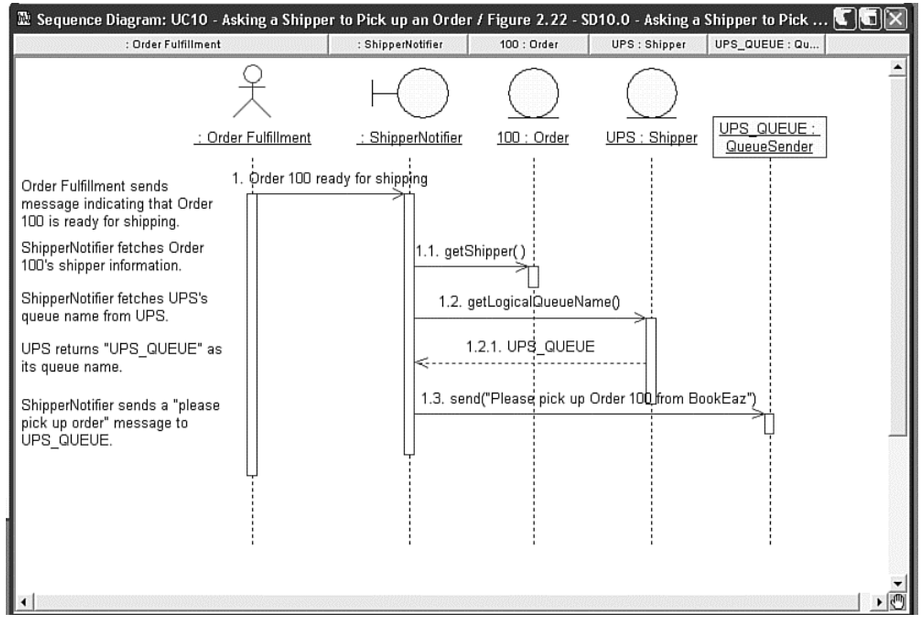


Figure 2.22

ShipperNotifier alerts shippers that orders are ready for pickup.

Discoveries

As Figure 2.22 shows, the fulfillment system is capable of notifying the main BookEaz system that an order is ready to be shipped. Therefore, your job is to implement a "listener" to capture these notifications and forward them to the appropriate shipping company. When the word "listener" appears in a software component's description, it usually indicates that the component will be some sort of asynchronous message processor, and in the J2EE world, "asynchronous message processor" equates to a Message-Driven Bean. Therefore, `ShipperNotifier` is a Message-Driven Bean.

`ShipperNotifier` actually plays both the message consumer and message producer roles. As an MDB, it receives (that is, consumes) messages from the fulfillment system. Upon receipt of a message, however, it changes roles and sends (that is, produces) a message to a shipper. This dual role-playing is often observed in MDBs; they tend to receive messages, make slight adjustments to those messages, and then forward the messages to other message processors.

Figure 2.22 also reveals the existence of Shipper, which proves to be another Entity Bean (it's a wrapper around persistent data). Shipper instances house information about the shipping companies with which BookEaz does business. The Shipper Entity Bean and the ShipperNotifier MDB are added to the logical model, yielding Figure 2.23.

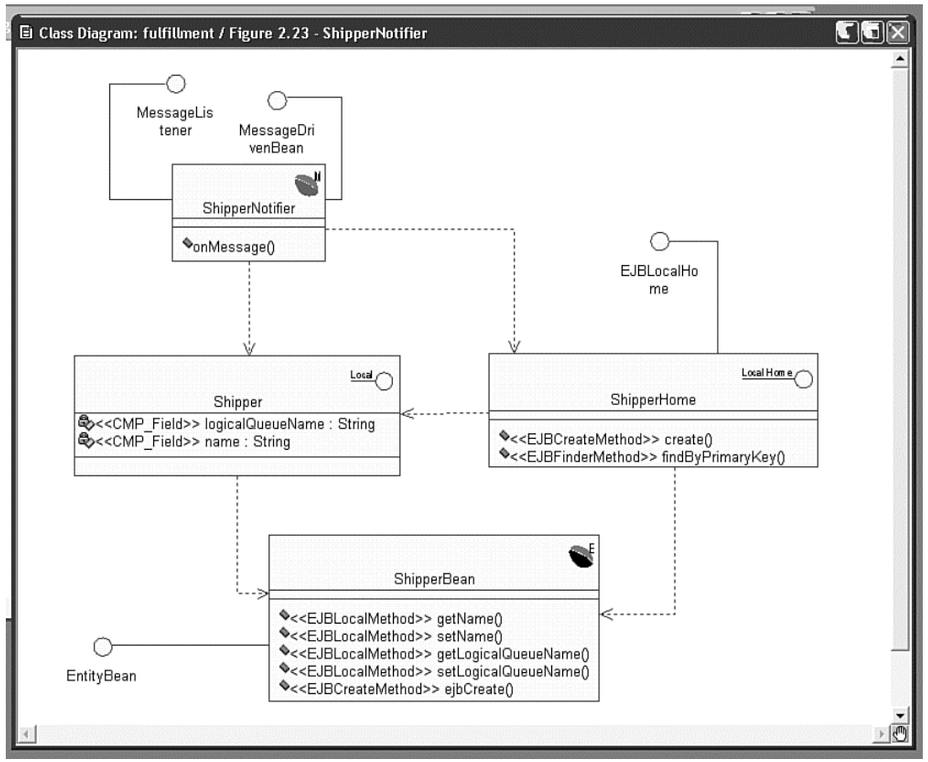


Figure 2.23

BookEaz communicates with shippers via the ShipperNotifier service.

SD11.0: Receiving Shipment Status Update

Customers who purchase items online have grown to expect up-to-the-minute reports about the whereabouts of their orders. All major shipping companies provide automated tracking systems that provide just this sort of information, and BookEaz intends to leverage those systems to keep their customers notified of their orders' travels. As shown in Figure 2.24, the ShipmentTracker component provides the means for shipping companies to keep BookEaz apprised of shipment statuses.

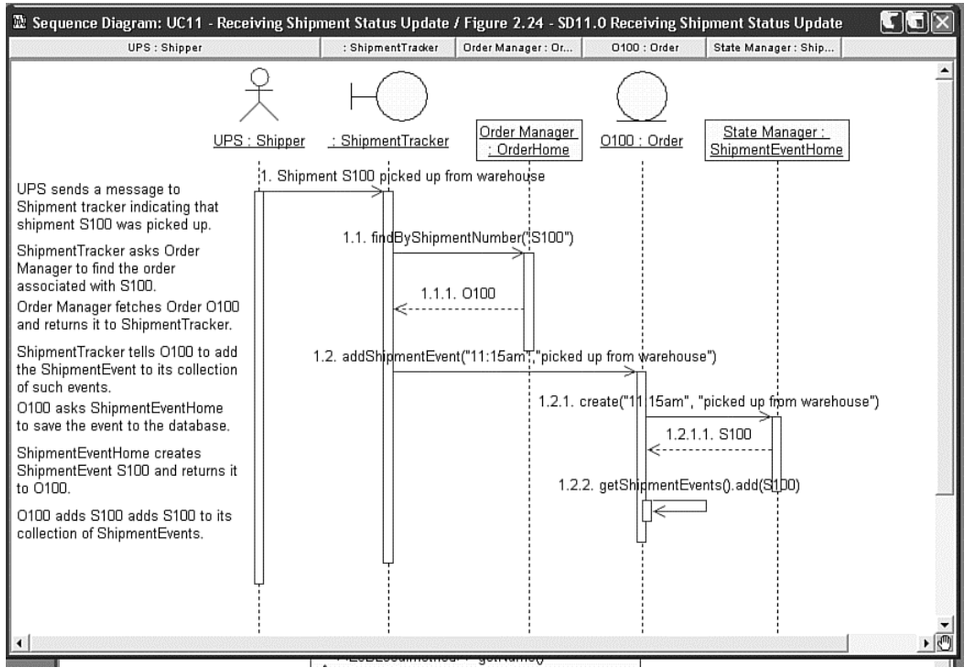


Figure 2.24

Shippers send shipment events to BookEaz via `ShipmentTracker`.

Shipping companies need to "fire and forget" messages concerning shipment status; that is, they need to communicate asynchronously with `ShipmentTracker`. As always, the presence of the word *asynchronously* strongly implies that an MDB is the preferred implementation; therefore, `ShipmentTracker` is a Message-Driven Bean.

Discoveries

Figure 2.24 reveals the existence of `ShipmentEvent`, which is BookEaz's last (for this release, anyway) Entity Bean. A `ShipmentEvent` holds two pieces of information: a description of a significant event, such as "shipment picked up from BookEaz warehouse," and the time that the event occurred. Figure 2.24 also indicates that `ShipmentEvents` are associated with `Orders`; this in turn implies that there is a container-managed relationship between `Order` and `ShipmentEvent`. Finally, this diagram demonstrates that shippers use their own internal tracking numbers when describing shipping events. This means that BookEaz needs a way to map a shipper's tracking number to an internal order number. This is accomplished by adding the `findByShipmentNumber` method to `OrderHome`, as shown in Figure 2.25.

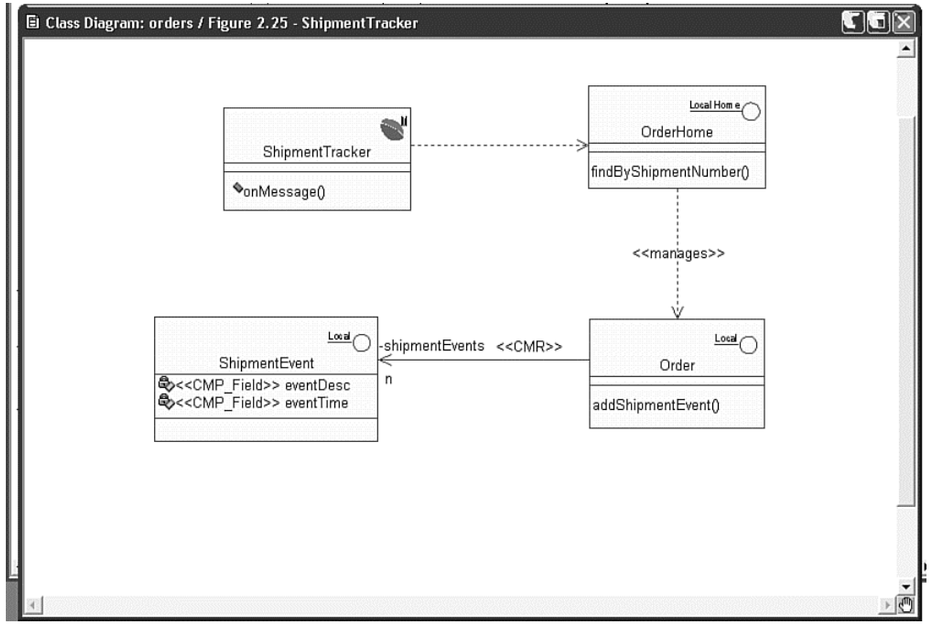


Figure 2.25

ShipmentTracker helps map ShipmentEvents to Orders.

Page 55, Chapter 3 opener: **Implementing a Solution Using EJB** 2.0 Now that you have documented the basic design tenets of the BookEaz system in the logical model, it is time to enter coding mode: The days of implementation have arrived. Although some of the Java code you'll see looks like normal J2SE code, much of it does not; EJB **2.0** radically and irrevocably alters the fundamental look and feel of Java code. The most pronounced change is in the quantity of Java code needed to implement a Java class. When transformed into EJB 2.0 components, J2SE-based classes containing hundreds of lines of code become EJBs containing only dozens of code	**Implementing a Solution Using EJB** 2.1 Now that you have documented the basic design tenets of the BookEaz system in the logical model, it is time to enter coding mode: The days of implementation have arrived. Although some of the Java code you'll see looks like normal J2SE code, much of it does not; EJB **2.1** radically and irrevocably alters the fundamental look and feel of Java code. The most pronounced change is in the quantity of Java code needed to implement a Java class. When transformed into EJB 2.0 components, J2SE-based classes containing hundreds of lines of code become EJBs containing only dozens of code lines. This

lines of code become EJBs containing only dozens of code lines. This dramatic reduction in source lines of code (SLOC) is caused by EJB **2.1** assuming many of the responsibilities formerly thrust upon implementers.	dramatic reduction in source lines of code (SLOC) is caused by EJB **2.1** assuming many of the responsibilities formerly thrust upon implementers.
Page 56, the Note: This chapter makes heavy use of **the J2EE Reference Server, a free (yet charmingly full-featured) EJB container.** Please follow the steps outlined in Appendix B, "Installing and Configuring the J2EE Reference Server," before proceeding with the code examples in this chapter.	This chapter makes heavy use of WebLogic Server, version 8.1. Please follow the steps outlined in Appendix B, "Installing and Configuring the J2EE Reference Server," before proceeding with the code examples in this chapter.
Page 58, paragraphs just below Figure 3.2 Every application server vendor prescribes a process for deploying EJBs. The examples in this book are based on **Sun's J2EE Reference Application Server, which is available free at http://www.javasoft.com**. I chose it for two reasons: **It is free, and it has a good deployment tool. I find deploying EJBs to the Reference Application Server much easier than deploying them anywhere else**. The BookEaz EJBs will deploy into any EJB **2.0**-compliant server, so you can use any one you choose. The next few pages of this book are specific to the Reference Application Server, however, so you might find it easier to follow along if you "go with the flow" and use the same application server. Appendix B supplies specific directions for installing and	Every application server vendor prescribes a process for deploying EJBs. The examples in this book are based on BEA's WebLogic Server; you can obtain an evaluation copy of it, complete with a one-year license, at www.bea.com. I chose it for two reasons, its market dominance and its adherence to standards. As developers, we need to ensure that we hitch our wagons to relevant technologies; learning about a product or paradigm that's not widely used is a waste of time. WebLogic is definitely the most widely used EJB container, at least in production-grade systems, so being familiar with this platform will definitely boost your attractiveness to potential employers and clients.

configuring **the Reference Application Server** and for getting a database connected to the application server. After installing the application server and configuring it per Appendix B's directions, you are ready to deploy your first EJB to it.	employers and clients. WebLogic's popularity can be attributed to many things (it's reliable, scalable, and enjoys great customer service, to name just a few of its attributes), but one of the most important is that it fully adheres to the EJB 2.1 standard. At the time of this writing, in fact, it is the only fully EJB 2.1-compliant commercial product. The BookEaz EJBs will deploy into any EJB 2.1-compliant server, so you can use any one you choose. However, this book is completely WebLogic-centric, so you might find it easier to follow along if you jump on the BEA bandwagon. Appendix B supplies specific directions for installing and configuring WebLogic and for getting a database connected to the application server. After installing the application server and configuring it per Appendix B's directions, you are ready to deploy your first EJB to it.
Page 59, second paragraph: UC6's functionality is manifested by the interaction of `Book`, an Entity Bean, and **BookAdmin**, a classic J2SE Java client class. Implementing `Book` introduces many of the fundamental tenets of EJB component construction, and implementing **BookAdmin** introduces the essential mechanisms for accessing and using those components.	UC6's functionality is manifested by the interaction of `Book`, an Entity Bean, and **BookAdministrator**, a classic J2SE Java client class. Implementing `Book` introduces many of the fundamental tenets of EJB component construction, and implementing **BookAdministrator** introduces the essential mechanisms for accessing and using those components.
Page 59, paragraph above the bulleted list: An entity with no "special needs" in its database operations, getters, or setters can be implemented as a CMP Bean. As of EJB **2.0**, the number of special needs has been reduced to the point that almost all Entity Beans can be implemented as CMPs. The only special needs remaining are as follows:	An entity with no "special needs" in its database operations, getters, or setters can be implemented as a CMP Bean. As of EJB **2.1**, the number of special needs has been reduced to the point that almost all Entity Beans can be implemented as CMPs. The only special needs remaining are as follows:

Page 60, Rule of Thumb 11 *Rule of Thumb 11* Under EJB **2.0**, almost all Entity Beans can and should be implemented as CMP Beans.	 *Rule of Thumb 11* Under EJB **2.1**, almost all Entity Beans can and should be implemented as CMP Beans.
Page 60, paragraph after "Declaring Book's Remote Interface" The time has come to start writing some code. Book, the Entity Bean designed in Chapter 2, "Designing a Solution Using EJB," is made up of four items: the Book **remote** interface, the BookHome home interface, the BookBean implementation class, and the deployment descriptor. The task of implementing Book naturally falls into four subtasks, one for each item composing the Entity Bean. It seems sensible to start by coding the two interfaces—Book and BookHome—because they determine the shape and size of BookBean and the deployment descriptor (see Listing 3.1). Make sure you have installed **the J2EE Reference Application Server before you proceed with coding**.	The time has come to start writing some code. Book, the Entity Bean designed in Chapter 2, "Designing a Solution Using EJB," is made up of four items: the Book **component** interface, the BookHome home interface, the BookBean implementation class, and the deployment descriptor. The task of implementing Book naturally falls into four subtasks, one for each item comprising the Entity Bean. It seems sensible to start by coding the two interfaces—Book and BookHome—because they determine the shape and size of BookBean and the deployment descriptor (see Listing 3.1). Make sure you have **installed WebLogic 8.1 (or higher) before your proceed with coding**..

Page 60, first part of Listing 3.1:
 	
Listing 3.1 Book's Remote Interface
 	
`package com.bookeaz.books;`
`import javax.ejb.EJBObject;`
`import java.rmi.RemoteException;`
 	
`/**`
`* A Book offered for sale by BookEaz. This is the` **`remote`**
`* interface part of the home/`**`remote`**`/implementation triad.`
`*/`
	
	Listing 3.1 Book's Remote Interface
	
	`package com.bookeaz.books;`
	`import javax.ejb.EJBObject;`
	`import java.rmi.RemoteException;`
	
	`/**`
	`* A Book offered for sale by BookEaz. This is the` **`component`**
	`* interface part of the home/`**`component`**`/implementation triad.`
	`*/`

Page 61, paragraph below the "Discussion" heading: Book conforms to a number of rules mandated by the EJB **2.0** specification:	 Book conforms to a number of rules mandated by the EJB **2.1** specification:

Page 61, last two bulleted items: • Its methods are `public`. Neither Java nor the EJB spec allows `protected` or `private` methods to be declared, so `Book`'s methods are all declared `public`. • Its methods all declare their propensity to throw `javax.rmi.RemoteException`. `Book` (and all Entity Beans, except Local Entity Beans, which are discussed later) throws this exception if anything goes irrecoverably wrong during any method **implementations**.	• Its methods are `public`. Neither Java nor the EJB spec allows `protected` or `private` methods to be declared **in interfaces**, so `Book`'s methods are all declared `public`. • Its methods all declare their propensity to throw `javax.rmi.RemoteException`. `Book` (and all Entity Beans, except Local Entity Beans, which are discussed later) throws this exception if anything goes irrecoverably wrong during any method **invocations**.
Page 62, paragraph above Rule of Thumb 12 In another respect, however, `Book` is an atypical Entity Bean remote interface. Most remote interfaces provide both a getter and a setter method for each attribute. `Book`, on the other hand, contains a getter for each of its attributes, but only one setter, the `setPrice()` method. `Book` is a "read mostly" Entity Bean; after a `Book` is created, there is rarely a need to change its information (such as `ISBN`, `author`, `subject`, or `title`), so these methods are intentionally removed from the **remote** interface. Because `Book` prices are subject to change, however, the `Book` **remote** interface provides a means for setting **the price attribute**. *Rule of Thumb 12* An Entity Bean attribute can be made "read-only" by simply excluding its setter method from the Bean's **remote** interface.	In another respect, however, `Book` is an atypical Entity Bean remote interface. Most remote interfaces provide both a getter and a setter method for each attribute. `Book`, on the other hand, contains a getter for each of its attributes, but only one setter, the `setPrice()` method. `Book` is a "read mostly" Entity Bean; after a `Book` is created, there is rarely a need to change its information (such as `ISBN`, `author`, `subject`, and `title`), so these methods are intentionally removed from the **component** interface. Because `Book` prices are subject to change, however, the `Book` **component** interface provides a means for setting **this particular attribute**. *Rule of Thumb 12* An Entity Bean attribute can be made "read-only" by simply excluding its setter method from the Bean's **component** interface.
Page 64, second bulleted point: • Home interfaces can declare any number of methods for creating new Entity Bean instances. These methods, which must have names beginning with `create`, are the EJB equivalent of constructors. `BookHome`'s single `create...()` method accepts values for all the attributes of the `Book` being created. `Create...()` methods return **remote** interfaces to the Entity Bean	• Home interfaces can declare any number of methods for creating new Entity Bean instances. These methods, which must have names beginning with `create`, are the EJB equivalent of constructors. `BookHome`'s single `create...()` method accepts values for all the attributes of the `Book` being created. `Create...()` methods return **component** interfaces to the Entity Bean instances they create.

instances they create.	interfaces to the Entity Bean instances they create.
Page 64, last bulleted point: • Finder methods can return a single instance (as `findByPrimaryKey()` does), or they can return multiple instances. Finder methods that return a single instance return a **remote** interface, so `findByPrimaryKey()` returns a `Book` **remote** interface. Finder methods that return multiple instances return a `java.util.Collection` or a `java.util.Set` containing instances of the home interface's corresponding **remote** interface.	• Finder methods can return a single instance (as `findByPrimaryKey()` does), or they can return multiple instances. Finder methods that return a single instance return a **component** interface, so `findByPrimaryKey()` returns a `Book` **component** interface. Finder methods that return multiple instances return a `java.util.Collection` or a `java.util.Set` containing instances of the home interface's corresponding **component** interface.

Page 68, beginning with the fifth line of the code:
`/**Methods that must be declared in order to conform`
`* to the EJB 2.0 spec. BookBean doesn't actually use`
`* these methods, however.`
	`/**Methods that must be declared in order to conform`
	`* to the EJB 2.1 spec. BookBean doesn't actually use`
	`* these methods, however.`

Page 69, paragraph after the "Discussion" heading: `BookBean` is a fairly typical CMP Entity Bean implementation class. It contains abstract getter and setter methods corresponding to the getters and setters declared in the `Book` **remote** interface. The EJB specification insists that getters and setters for CMP Entity Beans be abstract; the EJB container will provide concrete implementations for them on your behalf. It also contains a number of methods stipulated by the EJB **2.0** specification. Most of these methods contain empty bodies, a trait common in CMP Entity Beans; these empty implementations are subsumed by auto-generated methods supplied by EJB.	`BookBean` is a fairly typical CMP Entity Bean implementation class. It contains abstract getter and setter methods corresponding to the getters and setters declared in the `Book` **component** interface. The EJB specification insists that getters and setters for CMP Entity Beans be abstract; the EJB container will provide concrete implementations for them on your behalf. It also contains a number of methods stipulated by the EJB **2.1** specification. Most of these methods contain empty bodies, a trait common in CMP Entity Beans; these empty implementations are subsumed by auto-generated methods supplied by EJB.
Page 69, last bulleted item on page: • Its getters and setters are all abstract, as mandated by the EJB **2.0** specification for CMP Entity Beans. EJB graciously implements these methods on the developer's behalf.	• Its getters and setters are all abstract, as mandated by the EJB **2.1** specification for CMP Entity Beans. EJB graciously implements these methods on the developer's behalf.

developer's behalf.	
Page 70, first two bulleted items: • In contrast to the Book **remote** interface, BookBean contains the full complement of getter/setter pairs. The setters for ISBN, title, author, and subject are not exposed to the general public (via Book), but BookBean uses them internally to manipulate their respective attributes. • All of BookBean's getters and setters have public visibility, regardless of whether they are **visible to** the outside world. The EJB **2.0** specification insists that all getters and setters be public, even if they are not to be used outside the Entity Bean's implementation. The visibility of getters and setters is determined by their presence or absence in the **remote** interface, not by their visibility (public, private, "package," or protected) declaration in the implementation.	• In contrast to the Book **component** interface, BookBean contains the full complement of getter/setter pairs. The setters for ISBN, title, author, and subject are not exposed to the general public (via Book), but BookBean uses them internally to manipulate their respective attributes. • All of BookBean's getters and setters have public visibility, regardless of whether they are **meant to be used by** the outside world. The EJB **2.0** specification insists that all getters and setters be public, even if they are not to be used outside the Entity Bean's implementation. The visibility of getters and setters is determined by their presence or absence in the **component** interface, not by their visibility (public, private, "package," or protected) declaration in the implementation.
Page 70, the section from "Prerequisites for Compiling" at the bottom of the page, through page 71: To compile and run BookEaz code, your environment must be properly set up. Specifically, you need three software suites installed: • A Java Development Kit (JDK), version 1.3.1 or later, available at http://www.javasoft.com • Sun's J2EE Reference Server, available at http://www.javasoft.com and set up according to Appendix B • CloudScape's database system, which is bundled with Sun's J2EE Reference Server	To compile and run BookEaz code, your environment must be properly set up. Specifically, you need four software suites installed: • A Java Development Kit (JDK), version 1.4 or later, available at http://www.javasoft.com (WebLogic server also contains a copy of JDK 1.4.) • The ANT java-compilation system, available at www.jakarta.org (like the JDK, ANT is bundled with WebLogic Server). Refer to Appendix C for information regarding ANT. • BEA's WebLogicServer, version 8.1 or higher, available

Sun's J2EE Reference Server

In addition, you must have these environment variables set up for the compilation to proceed:

- CLASSPATH must contain a reference to j2ee.jar (which is part of Sun's J2EE Reference Server) and BookEaz's classes directory (<BOOKEAZ_HOME>/classes). To add them to your CLASSPATH on Windows systems, issue this command from a command-prompt window:

  ```
  set CLASSPATH=%CLASSPATH%;c:\j2ee\lib\j2ee.jar;c:\bookeaz\classes
  ```

- PATH must contain a reference to the JDK bin directory. To add this reference to PATH on Windows systems, issue this command from a command-prompt window:

  ```
  set PATH=%PATH%;<JAVA_HOME>\bin
  ```

  <JAVA_HOME> is the topmost directory where JDK was installed; for example, on my computer, <JAVA_HOME> is c:\jdk1.3.1.

During the post-tutorial parts of this book, you will use a modern build tool named Ant to compile and deploy EJBs. In the interest of getting something compiled and running quickly, however, compile Book.java, BookHome.java, and BookBean.java by hand for now. Follow these steps to compile Book's constituents on a Windows-based system:

1. Open a command-prompt window from the taskbar.

2. Change directory to <BOOKEAZ_HOME>\src\com\bookeaz\books (<BOOKEAZ_HOME> is the root of BookEaz's installation directory, typically c:\bookeaz).

3. Compile all three EJB components by issuing javac -d c:\bookeaz\classes Book*.java. This command compiles Book's components and places the resulting .class

at http://www.bea.com and set up according to Appendix B.

- Pointbase's database system, which is bundled with BEA's WebLogic Server. You must also build BookEaz's database per the instructions in Appendix B.

If you faithfully follow Appendix B's installation instructions, the following environment variables, which are required in order to successfully build BookEaz's software, will be set:

- BEA_HOME, which indicates the directory in which BEA products (most notably WebLogic Server) are located, will be set to c:\bea_81.

- WL_HOME, which indicates the directory that houses WebLogic Server, will be set to c:\bea_81\weblogic81.

In addition, you must have these environment variables set up for the compilation to proceed:

- CLASSPATH must contain a reference to weblogic.jar (which is part of BEA's WebLogic Server) and BookEaz's classes directory (<BOOKEAZ_HOME>/classes).

- PATH must contain a reference to the JDK bin directory.

The easiest way to set the CLASSPATH and PATH environment variables is to use setEnv.cmd, which resides in the <BOOKEAZ_HOME> directory.

You will use a modern build tool named ANT to compile and deploy EJBs. ANT is described in detail in Appendix C; in the interest of getting something compiled and running quickly, however, just invoke Ant via these instructions:

1. Open a command-prompt window from the taskbar.

2. Change directory to <BOOKEAZ_HOME>\src\com\bookeaz (<BOOKEAZ_HOME> is the root of BookEaz's installation

files into <BOOKEAZ_HOME>\classes\com\bookeaz\books.

After performing these steps, the <BOOKEAZ_HOME>\classes directory contains compiled versions of Book's three Java source files. You are now ready to deploy Book to the Application Server.

directory, typically c:\bookeaz).

3. Compile all three Book components (Book.java, BookHome.java, BookBean.java) by issuing ant compile_book. This command compiles Book's components and places them in a jar file.

When you issue the ANT command shown above, ANT examines the build.xml file and finds the following target:

```
<target name="compile_book" >
    <javac srcdir="${source}" destdir="${build}"
        includes=
        "books/Book.java,books/BookHome.java,books/BookBean.java"
     />
</target>
```

This target instructs ANT to compile Book's three constituent Java files when the compile_book target is invoked.

After performing these steps, you are ready to temporarily leave the comforts of the Java world for a foray into the world of deployment descriptors.

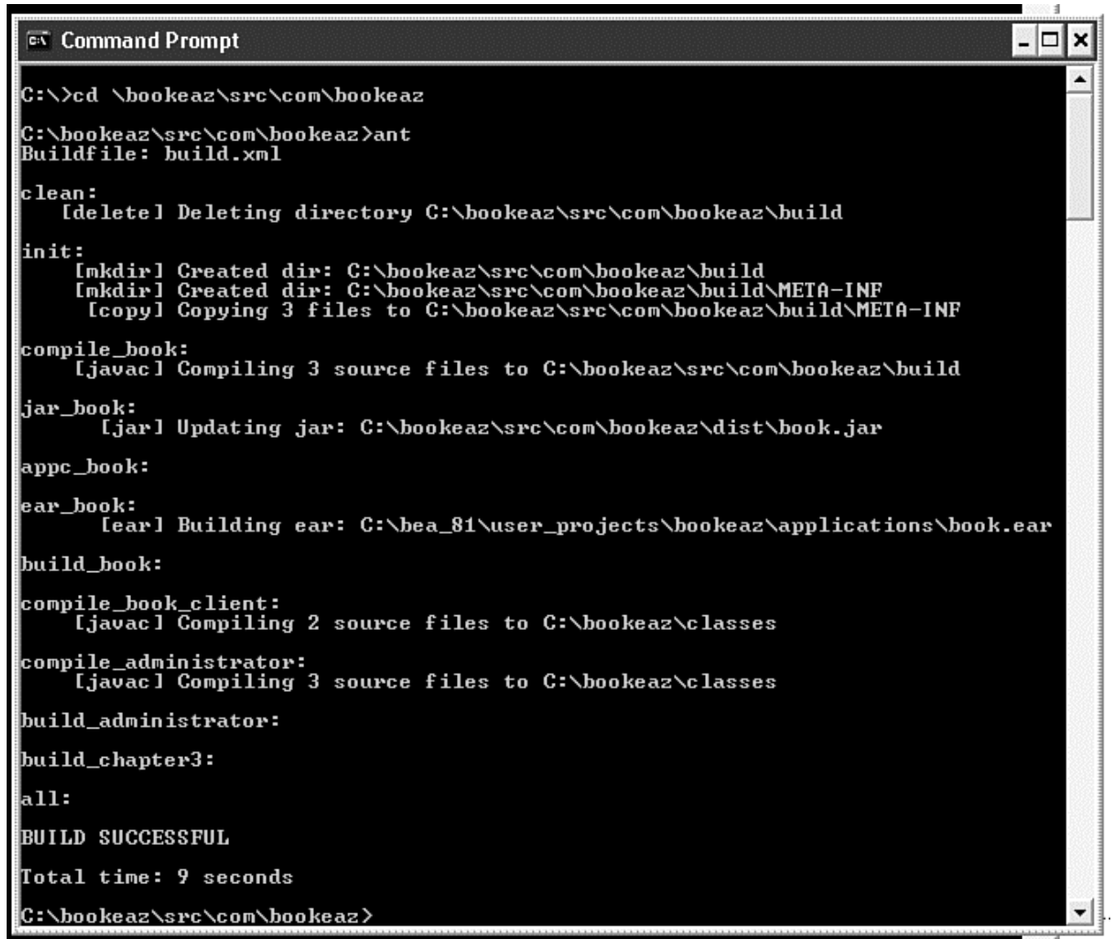


Figure 3.3

Compiling Book's code is easy when you use ANT to manage the process.

Page 73, two paragraphs just above the "Using deploytool to Deploy Book" heading: That's the bad news. Now here's the good news: Utility programs, commonly called "deployment tools," are available that can actually generate deployment descriptors for you. These tools, although not perfect, perform admirably enough that they largely eradicate the burden of creating deployment descriptors by hand. Sun's Reference Server is bundled with a handy deployment tool named, simply enough, `deploytool`. With `deploytool`, the `Book` EJB can be deployed quickly to the Sun Reference Server.	That's the bad news. Now here's the good news: Utility programs, commonly called "deployment tools," are available that can actually generate deployment descriptors for you. These tools, although not perfect, perform admirably enough that they largely eradicate the burden of creating **and maintaining** deployment descriptors by hand. **WebLogic Server is bundled with a deployment tool named WebLogic Builder. WebLogic Builder does a fine job of editing existing deployment descriptors, but sadly (and strangely, in my opinion), it is incapable of creating a new deployment descriptor. Fortunately, there is an open-source product named Perennial that can generate deployment descriptors for you. Perennial is an add-in to IBM's Rational Rose UML modeling. It allows Rose to generate fully formed EJB 2.1 deployment descriptors directly from UML class diagrams.** **Appendix D describes how to use Perennial to generate EJB 2.1 deployment descriptors. For the duration of part I of this book, however, you will be working with deployment descriptors that I have already created on your behalf, and you will be using ANT to deploy (that is, install) Book into your WebLogic server. Part II of this book contains a detailed reference document describing everything that can be configured in a WebLogic deployment descriptor.**
Bottom of page 73, "Using deploytool to Deploy Book," through the middle of page 78, "BookEaz's Database Administrator Client," replace all the text with the following:	

Using ANT to Deploy Book

ANT cannot help create deployment descriptor files, nor can it create any of the other configuration files needed by WebLogic; that must still be done by hand or via a tool such as Perennial. However, ANT does make deployment a simple

process. Once you create Book's deployment descriptor and configuration files (or obtain them from a willing source such as me), deploying Book becomes as simple as issuing a single ANT command.

The next few pages present a very high-level overview of the files that compose Book's deployment descriptor. Fortunately, you are spared the excruciating details of deployment descriptors until Chapter 6; for now this "50,000 foot view" will be sufficient grounding to get Book deployed and running.

The Application.xml File

The standard unit of deployment in WebLogic is the *application*. An application is a collection of J2SE components (plain old Java classes), J2EE components (for example, EJBs), and Web components such as JSPs and HTML files. An application is described by a manifest file named application.xml. Application.xml is essentially a list of the J2SE, J2EE, and Web components that make up an application. Book's application.xml file looks like this:

```
<application>

  <display-name>Book Entity Bean</display-name>
  <description>Book CMP Entity Bean as built in Chapter 3 </description>
  <module>
    <ejb>book.jar</ejb>
  </module>
</application>
```

Application.xml co-resides with build.xml in <BOOKEAZ_HOME>\src\com\bookeaz. The BookEaz source code, which you presumably have already acquired and installed on your computer, contains a copy of this file, so you don't have to create it by hand.

The ejb-jar.xml File

Entity Bean deployment descriptors are splayed across three separate files: ejb-jar.xml, weblogic-ejb-jar.xml, and weblogic-cmp-rdbms-jar.xml. Each file contains its own particular brand of deployment information. The ejb-jar.xml file houses basic structural information, including the following:

- The Entity Bean's name is declared in the <ejb-name> tag.
- The fully qualified names of the Entity Bean's component interface, home interface, and implementation class are declared via the <remote> (or <local>), <home>, and <ejb-class> tags.
- The class of the entity's primary key is declared via the <prim-key-class> tag. Book's primary key is the ISBN attribute, so its corresponding <prim-key-class> is declared to be of type String.

- Every container-managed attribute is declared via a <cmp-field> tag. These tags tell the container that it is responsible for generating get() and set() methods for the attributes, and for generating JDBC code for moving the attributes into and out of the database.

- If the entity has a simple primary key (that is, a primary key comprised of exactly one attribute), then that attribute is declared via the <primkey-field> tag. As you will see in later chapters, this tag is omitted for entities that use compound primary keys.

An elided snapshot of Book's ejb-jar.xml (the complete file is contained in the <BOOKEAZ_HOME>/src/com/bookeaz directory) is shown in Listing 3.4; there are numerous other pieces of vital information in ejb-jar.xml, but these are the only ones that are germane to the discussion at this point. Don't worry, though—you will be delving much more deeply into ejb-jar.xml throughout the remainder of this book.

Listing 3.4 Book's ejb-jar.xml File Declares Basic Structural Information

```
<entity>

<ejb-name>BookBean</ejb-name>
<home>com.bookeaz.books.BookHome</home>
<remote>com.bookeaz.books.Book</remote>
<ejb-class>com.bookeaz.books.BookBean</ejb-class>
<persistence-type>Container</persistence-type>
<prim-key-class>java.lang.String</prim-key-class>
<reentrant>False</reentrant>
<cmp-version>2.x</cmp-version>
<abstract-schema-name>BookSchema</abstract-schema-name>
<cmp-field>
    <field-name>iSBN</field-name>
</cmp-field>
<cmp-field>
    <field-name>title</field-name>
</cmp-field>
<cmp-field>
    <field-name>author</field-name>
</cmp-field>
<cmp-field>
    <field-name>subject</field-name>
</cmp-field>
<cmp-field>
    <field-name>price</field-name>
```

```
</cmp-field>
<primkey-field>iSBN</primkey-field>
```

The weblogic-ejb-jar.xml File

The weblogic-ejb-jar.xml file describes an EJB's (or a group of EJBs—one deployment descriptor can describe any number of EJBs) basic housekeeping and lifecycle-management characteristics. The following are the most notable settings in this file:

- The <entity-cache> stanza describes how many instances of the entity can be cached in the server's memory. Because BookEaz anticipates many users simultaneously manipulating many books, this number is set very high in Book's deployment descriptor.

- The <jndi-name> tag assigns a logical name to the entity's home interface. Book's clients pass this logical name to JNDI's lookup() method to obtain a copy of BookHome (more about this later in the chapter).

Weblogic-ejb-jar.xml can contain many other settings, but it is usually fairly small. Listing 3.5, which contains Book's weblogic-ejb-jar.xml file, is a typical rendition of this file.

Listing 3.5 Book's weblogic-ejb-jar.xml File

```
<weblogic-ejb-jar>

<weblogic-enterprise-bean>
    <ejb-name>BookBean</ejb-name>
    <entity-descriptor>
        <entity-cache>
            <max-beans-in-cache>1000</max-beans-in-cache>
        </entity-cache>
        <persistence>
            <persistence-type>
                <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
                <type-version>6.0</type-version>
                <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml</type-storage>
            </persistence-type>
            <persistence-use>
                <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
                <type-version>6.0</type-version>
            </persistence-use>
        </persistence>
```

```
    </entity-descriptor>
    <jndi-name>BookHome</jndi-name>
</weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

The weblogic-cmp-rdbms-jar.xml File

The final file comprising Book's deployment descriptor goes by the long-winded name weblogic-cmp-rdbms-jar.xml. This file proves the adage that the length of a file's name is directly proportional to the importance of its contents, for this deployment descriptor file contains the information that WebLogic uses to map Book's attributes to the database table in which they are stored. The following are the primary deployment directives contained in this file:

- The <table-name> element indicates the database table that houses the entity's instances. Book, for example, is mapped to the BOOKS database table via this element.
- The <data-source-name> element indicates the physical database instance that houses the database table mentioned in the <table-name> element. (More formally, it indicates the DataSource from which the EJB container acquires connections to the underlying database instance; more about this in Appendix B.)
- Each container-managed attribute is mapped to a database column via a <field-map> stanza. Book's price attribute, for example, is mapped to the column named BOOK_PRICE.

Book's weblogic-cmp-rdbms-jar.xml file, like all of Book's deployment descriptor files, is contained in the <BOOKEAZ_HOME>/src/com/bookeaz directory.

Listing 3.6 Book's weblogic-cmp-rdbms-jar.xml File Describes How to Map the Entity's Attributes to Its Associated Database Table

```
<weblogic-rdbms-jar>

    <weblogic-rdbms-bean>
        <ejb-name>BookBean</ejb-name>
        <data-source-name>BookEazDataSource</data-source-name>
        <table-name>BOOKS</table-name>
        <field-map>
            <cmp-field>iSBN</cmp-field>
            <dbms-column>ISBN</dbms-column>
        </field-map>
        <field-map>
            <cmp-field>title</cmp-field>
```

```
            <dbms-column>BOOK_TITLE</dbms-column>
        </field-map>
        <field-map>
            <cmp-field>author</cmp-field>
            <dbms-column>BOOK_AUTHOR</dbms-column>
        </field-map>
        <field-map>
            <cmp-field>subject</cmp-field>
            <dbms-column>BOOK_SUBJECT</dbms-column>
        </field-map>
        <field-map>
            <cmp-field>price</cmp-field>
            <dbms-column>BOOK_PRICE</dbms-column>
        </field-map>
    </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

Deploying Book to WebLogic

Now that `Book`'s code, deployment descriptor, and `application.xml` file have been created, it's ready to be deployed into WebLogic. Deploying `Book` is now merely a matter of executing an ANT target, which can be done by following these steps:

1. Open a command-prompt window.

2. CD to `<BOOKEAZ_HOME>\src\com\bookeaz`.

3. Set your `CLASSPATH` and `PATH` by issuing the `\bookeaz\setEnv` command.

4. Issue the ANT `ear_book` command, which tells ANT to execute the commands listed in `build.xml`'s `ear_book` target.

As a result of these actions, which are depicted in Figure 3.4, an EAR file, `book.ear`, is created and deposited into WebLogic's applications directory. `Book` is now said to be "deployed to WebLogic", which means that its services can be used by any client with access to WebLogic's EJB facilities.

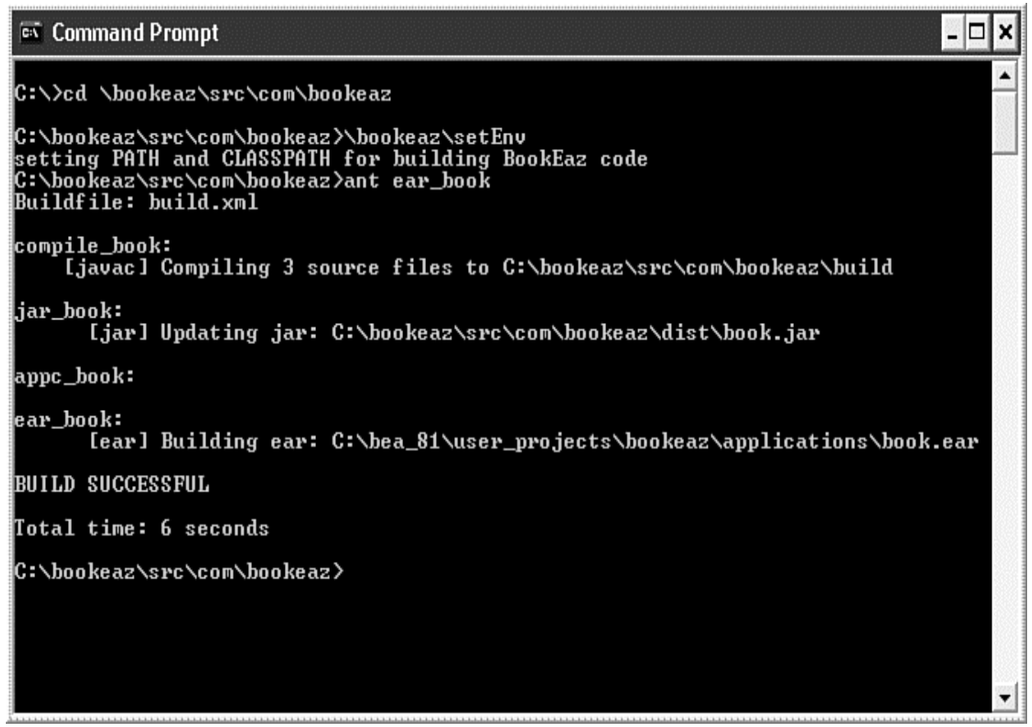


Figure 3.4

The ear_book ANT target assembles and deploys Book's enterprise application archive (EAR) file to WebLogic.

Congratulations! You have just deployed your first Entity Bean. `Book` is now ready to participate in UC6. All you need now is a client program to interact with the Entity Bean.

Page 78, first item of the bulleted list at the bottom of the page, change to the following: • It establishes contact with WebLogic's JNDI (Java Naming and Directory Service)by acquiring an object called an `InitialContext`. JNDI is the repository from which clients obtain (among other things) EJB home interface instances:	

`//establish contact with the Book Entity Bean`
`//via the wonders of java naming (JNDI)`
`//this is how one obtains a home when one is NOT`
`//running in the same JVM as the EJB`
`Properties h = new Properties();`
 	
`h.put(Context.INITIAL_CONTEXT_FACTORY,`
`"weblogic.jndi.WLInitialContextFactory");`
 	
`h.put(Context.PROVIDER_URL, url);`
`InitialContext ctx = new InitialContext(h);`	

Page 79 through the Conclusion on page 80, change to the following text:	

Thanks to ANT, compiling `BookAdministrator` is relatively easy:

1. Open a command-prompt window.
2. Change the directory to `<BOOKEAZ_HOME>\src\com\bookeaz\`.
3. Set the `PATH` and `CLASSPATH` via the `\bookeaz\setEnv` command.
4. Compile `BookAdministrator` and its accompanying exception classes by issuing this command:

   `ant build_administrator_`

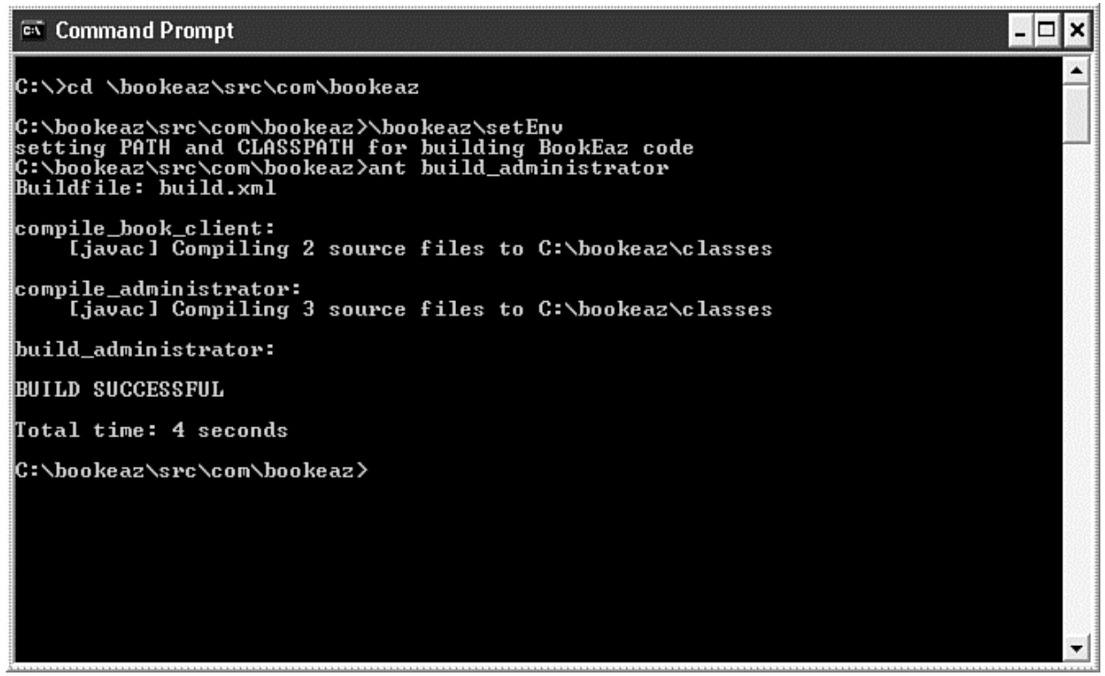


Figure 3.5

ANT's build_administrator target compiles BookAdministrator.

The `build_administrator` ANT target performs the following actions:

- It compiles `com.bookeaz.admin.BookAdminstrator` and places the resulting `.class` file into the `<BOOKEAZ_HOME>/classes` directory structure.
- It compiles `BookHome` and `Book` and places their `.class` files into the `<BOOKEAZ_HOME>/classes` directory structure.

It might seem odd that you have to compile `BookHome` and `Book` again—after all, they were already compiled when the `Book` EAR file was built and deployed. This apparently duplicate effort is standard procedure in the EJB world, however; one has to assume that a client program such as `BookAdministrator` has no access to EAR files that have been deployed to WebLogic. Therefore, an EJB's publicly visible interfaces must always be compiled again so that they are available to clients.

Running BookAdministrator is scarcely more difficult than compiling it. The only oddity in the process is that a new directory, <BOOKEAZ_HOME>/classes, must be added to CLASSPATH for BookAdministrator to work properly.

Follow these steps to run BookAdministrator:

1. Open a new command-prompt window (or reuse an already-opened one).

2. Use the task bar to start WebLogic (Appendix B contains detailed instructions for starting WebLogic Server).

3. Add <BOOKEAZ_HOME>\classes to CLASSPATH by issuing the following command (substituting in your actual value of <BOOKEAZ_HOME>):

   ```
   <BOOKEAZ_HOME>\setClientEnv.cmd
   ```

4. Change the directory to <BOOKEAZ_HOME>.

5. Start BookAdministrator with this command:

   ```
   java com.bookeaz.admin.BookAdministrator books.dat
   ```

You just used BookAdministrator to establish BookEaz's initial inventory of Books! I normally refrain from using exclamation points (they ruin my facade of stoicism), but I felt it necessary to make an exception here because you have just used your first Entity Bean to accomplish a real task.

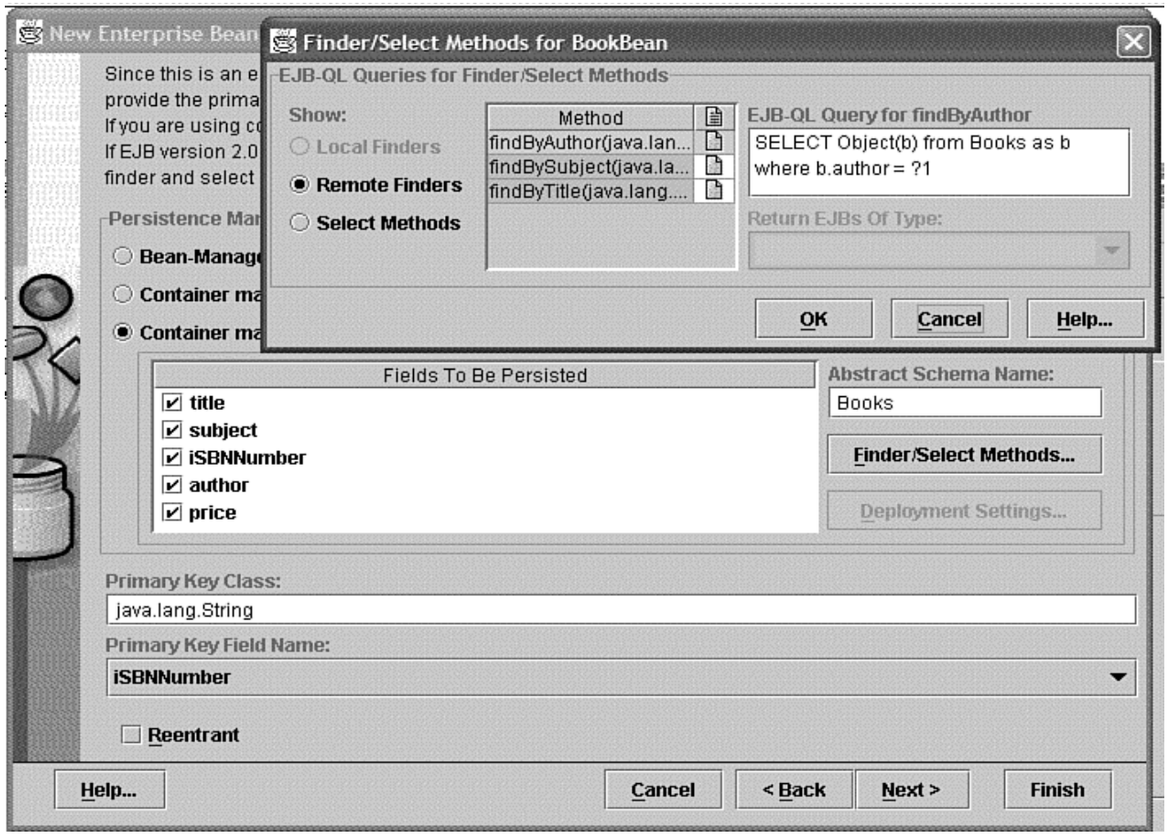


Figure 3.6

BookAdministrator embodies use case UC6's functionality.

Page 80, first part of last paragraph: The myriad details of deployment descriptors were glossed over in this chapter for two reasons. First, deployment tools, such as **deploytool**, are starting to make it unnecessary to delve too deeply into the details of deployment descriptors	The myriad details of deployment descriptors were glossed over in this chapter for two reasons. First, deployment tools, such as **WebLogic Builder and Perennial**, are starting to make it unnecessary to delve too deeply into the details of deployment descriptors

This errata sheet is intended to provide updated technical information. Spelling and grammar misprints are updated during the reprint process, but are not listed on this errata sheet.