CHAPTER 10

The Sample Application

TO conclude this discussion of the J2EE programming model, this chapter provides an in-depth description of a multitier Web application, an e-commerce Web site. We review the entire process of developing this application from specification to design to implementation, illustrating many of the principles discussed in the earlier chapters.

The first section describes some scenarios in which the sample application is used. Although the sample application supports administration and business-to-business interactions as well as shopping interactions, this chapter focuses mainly on shopping interactions.

The discussion then turns to the architecture of the sample application: the partitioning of functionality into modules, the assignment of functionality to tiers, and object decomposition within tiers. The architecture of the sample application conforms to the Model-View-Controller architecture. We describe the motivation for using this architecture and how each of these concepts is realized in the implementation of the sample application.

Finally, this chapter describes how the sample application uses the deployment, transaction, and security capabilities of the J2EE platform to simplify component development and provide richer functionality.

10.1 Application Functionality

The sample application models a typical e-commerce application, an online pet store. E-commerce sites like this are among the most common Web applications. The application interface is presented to its customers through a Web site and a customer interacts with the application using a Web browser. Other potential users of the application include administrators responsible for maintaining inventory and

performing other managerial tasks, and associated businesses such as suppliers. Each class of users would have access to specific categories of functionality, and each would interact with it through a specific user interface mechanism.

Like a typical e-commerce site, the pet store presents the customer with a catalog of products. The customer selects items of interest and places them in a shopping cart. When the customer has selected the desired items and indicates readiness to buy what is in the shopping cart, the sample application displays a bill of sale: a list of all selected items, a quantity for each item, the price of each item, and the total cost. The customer can revise or cancel the order. When the customer is ready to accept the order, the customer provides a credit card number to cover the costs and supplies a shipping address.

10.1.1 Scenarios

The following scenarios demonstrate a few key ways the pet store application could be used by describing a user's view of interactions with the system. By walking through these scenarios, you'll gain a better understanding of the requirements as well as the interactions that happen *within* the system.

The sample application could support three very different kind of scenarios. First, there is the shopping interface described earlier, that allows shoppers to buy items online. Second, there is an administration interface for carrying out store administration activities. Finally, there is a business-to-business interface through which the store can interact with suppliers. The scenarios in this section demonstrate all three types of interaction, while the remainder of this chapter focuses mainly on the shopping interactions.

10.1.1.1 Shopping Scenario

The primary function of the sample application is to provide an interface where customers can browse through and purchase items. This shopping interaction typically starts with the customer's visit to the application home page and ends when the customer orders from the site:

1. A customer connects to the application, by pointing the browser to the URL for the application's home page. This allows the customer to browse through the catalog or search for products through some search interface.
2. At any point during the whole interaction, the customer can sign into the application by providing an account identifier and a password. When the customer

signs in, the application can recall information about the customer such as a preferred shipping address and billing information, buying preferences, and so on. Customers who don't have an account can create one at any time by providing an account identifier, customer name, password and some other personal details.

3. The customer browses through the catalog. The customer can select a category to see a list of all the products in that category. For example, the customer can select the category `Cats` to view all cats that the pet store sells. Alternatively, the customer can search for products using one or more keywords describing the product. For example searching with keywords `Persian` and `mammal` might bring a list of Persian dogs and cats.

4. The customer selects a particular product in the list. Now, the application displays detailed information about the selected product. The description and image of the product is shown along with pricing information. When there are several variants of the same product, each variant is shown as a separate item. For example, when showing details about an African parakeet, the items could be large male African parakeet, small female African parakeet, and so on.

5. The customer decides to purchase a particular item and clicks a button to add the item to the shopping cart. The customer may continue shopping, adding more items to the shopping cart. As the customer browses through the catalog, the application remembers all the items placed in the cart. The customer can recall the shopping cart at any time during the interaction to review or revise the contents of the cart.

6. The customer can choose to order the items in the shopping cart at any time. This is called checking out. A checkout button is presented along with the shopping cart. If the customer is not signed in, the application brings up a signin/signup screen. Here the customer can sign in, or set up a new account, if they don't have one. After the customer is signed in, order processing continues as earlier.

7. When the customer asks to checkout, the application present a summary of all items that would be ordered along with their costs. At this point the customer must confirm the order.

8. When customer confirms the order, the application begins to gather shipping and billing information for the order. First it presents a form, where the customer can enter shipping information. If the customer is signed into the application at this time, the form comes up filled in with the customer's preferred shipping address.

9. When the customer enters the shipping address, the customer is asked to enter

billing information, including credit card details and a billing address. If the customer is signed in, the application recalls these details and the forms are returned filled in.

10. Finally the customer confirms the order and the application accepts the order for delivery. A receipt including a unique order number and other order details is presented to the customer. The application validates credit card and other information, updates its inventory database, and optionally sends a confirmation message via email.

This is a fairly typical shopping scenario. Some variations are possible, especially in the way the catalog is presented to the customer. For instance, the application could provide specialized lists of items such as best-sellers, or discounts on certain items. There may also be variations in order processing, such as reducing the steps for making an order when the customer is already signed in. The application developer needs to design the application to support these variations, as well as others that might arise as the application evolves.

Although this scenario presents the application from a single customer's point of view, the pet store application needs to simultaneously support a large number of shoppers.

10.1.1.2 Administration Scenario

The pet store application does most of the administrative work of managing orders, creating new accounts, and other details without manual intervention. However, there are some tasks where manual intervention is desirable or required. These are often administration tasks, such as managing the inventory, reestablishing forgotten customer passwords, rolling back orders, handling returned merchandise, and processing and shipping of orders.

The administration interface of the pet store application could use a Visual Basic client running in a Microsoft desktop application such as Microsoft Excel. The administration scenario models inventory management, where an administrator updates inventory when new shipments come in:

1. The administrator starts up the shopping client application. When the client starts, it asks the administrator to sign on to the system using a user name and password. The administrator enters information for one of the accounts that has administration privileges.

2. The client application then presents a list of products in the catalog, perhaps in

order by category, with the product details such as description and name also shown.

3. The administrator clicks a product to see the items as well as their inventory status. For any item displayed, the administrator can modify the inventory status.
4. The administrator clicks an update button, causing the changes to inventory status to be committed to the inventory database.

The application must be designed to support more than one administrator simultaneously using the administration interface.

10.1.1.3 Business-to-Business Scenario

Businesses often have a need to interact with other businesses through their custom applications. For example, a retailer needs to work with suppliers to procure inventory, with shipping agencies for managing shipments, and with billing agencies for handling its billing needs. In fact, significant pieces of the application such as inventory control could themselves be off-loaded to a separate business.

It would be desirable to have some of these interactions be automated. When businesses are tightly coordinated, perhaps under the same ownership or administration, these interactions could be *closely-coupled.* In such interactions, businesses expose their entities and data to each other. However, most of the time it is desirable to keep the businesses *loosely-coupled*. Here businesses interact by passing asynchronous messages to each other. This messaging approach also models the real world more closely, where businesses work together by sending faxes and packages, and so on, to each other.

An interaction between the pet store and one of its suppliers would illustrate a loosely-coupled business interaction. A typical scenario might be:

1. A customer places an order. This causes the inventory to fall below a pre-established low water mark, triggering the application to initiate an order to obtain more items from the supplier. This process happens asynchronously and does not interfere with the transaction being performed by the customer.
2. The application sends a purchase request message to the supplier. A typical purchase request message could say, “Send 100 male African parakeets.”
3. At some later time, the supplier sends a message in response to the request. If the supplier does not have enough parakeets to fill the order, the message might say, “Can’t fulfill request. Have 20 parakeets available.”

4. The application, upon receipt of the message, might send another request for a smaller quantity. The message might say, “Send 20 male African parakeets.”
5. The supplier initiates delivery of the shipment, and sends a message back to the application. This message might say, “Request completed. 20 parakeets shipped. Shipment number is 1234.”

The interaction between the store and supplier is depicted in a timing diagram in Figure 10.1.

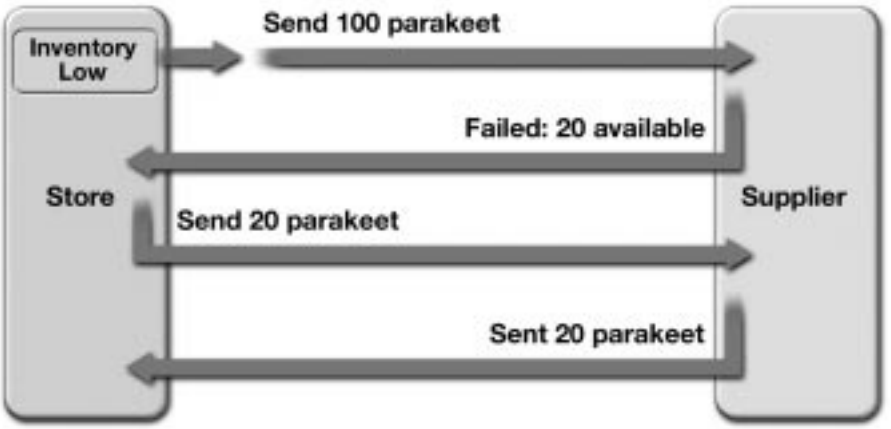

Figure 10.1 A Store-Supplier Business-to-Business Interaction

One thing to observe about this scenario that it is asynchronous. The action is initiated when a customer places an order. However, it proceeds without blocking the customer’s interaction. Also note that neither the store nor the supplier is blocked waiting for the other to respond. While the procurement is in progress, the store’s application and the supplier’s system carry on with their activities as usual.

10.1.2 Functional Specification

With a clear understanding of the kind of scenarios in which the application would be used, let’s create an initial specification of the user interface of the application. This section presents a sketch of the main user interface of an application that supports the shopping interactions. It is possible to create a similar sketch for a user interface for administration interactions. Business interactions typically do not

require a user interface. As mentioned earlier, the remainder of this chapter focuses on the shopping functionality of the application.

Upon arriving at the main page of the online pet store, a customer would expect some of the following features:

- A set of links or navigation bars on each page that provide quick access to common navigational tasks.
- An organized view of the site's contents through a categorized catalog.
- A search mechanism to provide a way to locate items based on keyword descriptions. Other types of quick access could be in terms of popular items or new additions.
- A *master view* of the catalog that lists items of interest. This could be the result of the customer navigating through a catalog category or the outcome of a keyword search.
- A *detail view* that describes the details of a particular item. Shoppers click on an item in the master view to zoom in on details, including a description, a picture, the price, a link to the supplier's URL, and so on.
- A shopping cart view that lets customers review the contents of their shopping cart. The cart allows the customer to modify quantities of items in the cart, including removing items from the cart altogether.
- A checkout or bill-of-sales view that displays the total order cost and allows the customer to enter billing and shipping information. The customer will want assurance that order details including shipping and credit card information are transferred securely and accountably. The interaction must be authenticated to positively identify the customer for the purposes of accountability and encrypted through HTTPS to protect the privacy of the information the customer provides.
- A receipt view to provide confirmation of the purchase through a unique order identifier or other mechanism to track the newly placed order and review details of the order.

In addition to these user interface requirements, the application must also support some security requirements. We address these in section 10.11 on page 309.

10.2 Application Architecture

This section describes the architecture of the pet store application; exploring the partitioning of functionality into modules, the assignment of functionality to tiers, and object decomposition within the tiers.

10.2.1 Application Modules

This discussion reviews the shopping interaction scenario once again, this time identifying actions within the application as it runs on the server. This replay is used to explore ways to divide the application into modules based on similar or related functionality. Dividing the problem in this manner reduces the dependency between modules, allowing them to be developed somewhat independently. Identifying the interface between modules enables some of the modules to be provided by third-party component providers, or subcontracted to specialists in a particular area of functionality.

Here's the scenario once again, with the various behaviors organized by modules:

1. A user connects to the application. If the user logs in, the *user account module* maintains user account information. It creates new user accounts and manages these accounts. Accounts include such information as user name, password, and account ID.
2. The *product catalog module* returns a list of products. The product catalog module searches the product database for a list of possible matches to the search criteria and renders the products for the user.
3. The user views a specific product. The product catalog module also returns detailed information about the selected product, including pricing information. It optionally can check the *inventory module* for availability information, such as quantity in stock.
4. The user selects an item for purchase. The *shopping client module* creates a shopping cart for the user for the duration of the user's session.
5. The user chooses the checkout option and commits to buying the item. The *order processing module* manages this interaction.
6. The application determines whether the user is logged in and if not, calls the user account module to set up a new user account. Otherwise it instructs the user account module to extract account information such as credit card and

shipping information.

7. The application then authenticates the user and validates the credit card and shipping information.
8. The application lets the user revise or cancel the order. If the user accepts the order, the order processing logic logs the order, notifies the *inventory module* to update the inventory database, and sends a confirmation message by email.

This time, the run-through of the scenario has identified the following modules and their responsibilities:

- User account module - The application tracks user account information. This includes a user identifier and password and various types (billing and email addresses, phone number, and so on) of contact information. The application saves user account information to a database so that it spans sessions.
- Product catalog module - The application allows the user to search for products or services and be able to display details of individual products. The product catalog includes descriptions of individual items.
- Order processing module - The application performs order processing. Order processing occurs when the user performs the check-out process and buys the items in the shopping cart.
- Messaging module - The application sends confirmation messages.
- Inventory module - The application maintains information on the number of each type of product in stock.
- Control module - The application allows users to browse the product catalog and add selected items to a shopping cart. At any time, the user can modify items in the shopping cart, add new items, or remove items already placed in the cart.

Figure 10.2 shows the interrelationship of the modules in the sample application.

Figure 10.2 Functional Modules for the Sample Application

This modular decomposition of the pet store application is reflected in the subpackages of the sample application's top-level package `com.sun.estore`:

- `account` - user account
- `cart` - shopping cart
- `catalog` - product catalog
- `control` - controls
- `inventory` - product inventory
- `mail` - email messaging
- `order` - order processing
- `util` - utility classes

10.2.2 Application Design

Partitioning the application into logical modules is the first step in subdividing the overall problem. The next step is to begin the process of object-oriented design of the application, identifying units of business logic, data, and presentation logic and modeling each of them as a software object.

The process starts by identifying the options and approaches available at the highest level. Once these choices are clear and the decisions and design principles are established, the rest of the design will be simplified by leveraging these overall principles.

One of the first decisions to make concerns the tiers that the application uses. The J2EE platform is designed for multitier applications, and offers a lot of flexibility in choosing how to distribute application functionality across the tiers. In a Web-enabled application, such as the sample application, some tiers are always present: the client tier provided by the browser, the Web tier provided by the server and the enterprise information system or database tier which holds persistent application data. The first choice to make is whether the Web tier accesses the enterprise information system resources directly, or goes through an EJB tier. The decision depends on the functionality, complexity, and scalability requirements of the application. Since such requirements can change as the application evolves, one goal for the design is to make it amenable to migration to an EJB-centric approach.

After deciding what tiers constitute the application, the next decision is how to distribute application functionality across these tiers. This division is closely linked to how the application is divided into objects at the highest level and represents one of the most important decisions when designing enterprise applications. Some clear and simple guidelines to help with making this decision are addressed in the following discussions.

10.2.2.1 Application Tiers

In a Web-centric design, the Web tier communicates directly with the enterprise information system resources that hold application data. In this approach, the Web tier is responsible for almost all of the application functionality. It must take care of dynamic content generation and presentation and handling of user requests. It must implement core application functionality such as order processing and enforce business rules defined by the application. Finally, the components running in the Web tier must also manage transactions and connection pooling for data access. Because it must handle so many functions, Web-centric application software has a tendency

to become monolithic. As a result, unless special efforts are taken, it does not scale well with increasing software complexity.

In an EJB-centric design, enterprise beans running on EJB servers encapsulate the enterprise information system resources and the core application logic. The Web tier communicates with the EJB tier instead of directly accessing the enterprise information system resources. This approach moves most of the core application functionality to the EJB tier, using the Web tier only as a front end for receiving client Web requests and for presenting HTML responses to the client.

The principal advantage of this approach is that enterprise beans have access to a broad set of enterprise-level services. Because of these services, managing transaction and security aspects of the application is easier. The EJB container provides a highly structured environment for the components that allows a developer to focus entirely on the application domain issues, while the EJB container takes care of system-level details. These standardized container-provided services also translate into better software reliability. The EJB architecture supports a programming discipline that promotes encapsulation and componentization, resulting in software that stays manageable as applications grow more complex.

The Web-centric approach is better for getting the application off to a quick start, while EJB-centric approach becomes more desirable when building a large scale application where code and performance scalability are prime factors. While the Web-centric approach may be more prevalent, with many applications implemented using it, it has limitations when building large scale, complex applications.

The ideal solution is an approach that benefits from the strengths of both approaches. The sample application demonstrates an approach that started out simple and small, but kept the option of growth open. Its extensible design started as Web-centric and migrated to an EJB-centric architecture. While most of its modules are implemented with an EJB-centric design, the catalog module uses the Web-centric model. Strategies for migrating components from Web-centric to EJB-centric designs are described in detail in section 4.7.1 on page 122.

Note that the discussion that follows describes a sample application design that evolves from Web-centric to EJB-centric, the actual code of the sample application reflects the final result of that migration. We have preserved the state of the catalog module before migration to provide an indication of how the migration was performed.

10.2.2.2 Application Objects

The next issue to address in developing the overall application architecture is how to subdivide the application into objects and how to the assign these objects to tiers. This process is referred to as object decomposition. While most of the objects are consigned to one tier or another, there are some that serve to connect the tiers and will need to span tiers, and their design needs to take this into account.

This discussion focuses primarily on large scale, complex applications. Smaller applications can probably get away with less rigorous treatment, but object design really becomes important as applications grow more complex. Large scale development of object-oriented software requires frameworks. It is important to have a framework, so that every time the design requires two objects to interact, a developer does not have to come up with a whole new notion of how the interaction works out.

This section looks at the issues to keep in mind when doing the object decomposition, and present techniques that we used in the sample application to determine an effective decomposition.

10.2.2.2.1 Design Goals

Consider the kind of goals that need to be addressed in object decomposition. Each of these considerations identifies criteria to use to divide the application. The framework must enable:

- Reuse of software designs and code
- Identification of responsibility of each object. The division into objects must ensure that the responsibilities of each object—what the object represents and what must it accomplish—are easily and unambiguously identified.

While these requirements apply to object-oriented design in general, they become even more important for multitier enterprise applications. Our additional objectives were:

- Separate of stable code from more volatile code. All parts of an enterprise application are not equally stable. The parts that deal with presentation and user interface change more often. The business rules and database schemas employed in the application have a much lower propensity to change. The overall architecture should separate stable portions of the application from parts that

are more volatile.

- Divide development effort along skill lines. The people that comprise an enterprise development team typically represent a very diverse set of skills. There are HTML layout and graphics designers, programmers, application domain experts, and enterprise information system resource access specialists, among others. The decomposition should result in a set of objects that can be assigned to various subteams based on their particular skills. This division of labor allows work on each object to proceed in parallel.

- Ease of migration from Web-centric to EJB-centric design. As mentioned earlier, the sample application starts out as a Web-centric application and migrate to being EJB-centric.

We have described these considerations from the point of view of a high-level division. However they are equally applicable even when we are working on identifying objects at a finer level. We will keep coming back to these considerations as we need to make choices about object decomposition.

10.2.2.2.2 MVC Architecture

When applying the considerations discussed above to the sample application, the first lines of division start becoming clear. At the highest level, the application divides into three logical categories of objects. These are objects that deal with *presentation* aspects of the application, objects that deal with the *business* rules and data, and objects that accept and interpret user requests and *control* the business objects to fulfill these request.

The look and feel of the application interface changes often, its behavior changes less frequently, and business data and rules are relatively stable. Thus objects responsible for control are often more stable than presentation objects while business rules and data are generally the most stable of all.

The implementation of presentation objects is typically handled by graphics designers, HTML and JSP technology experts, and application administrators after the application has been deployed. Control-related objects are implemented by application developers. Business rules and data objects are implemented by developers, domain experts, and database experts.

The presentation logic of a user interface can be handled by the Web tier or the client. In the Web tier, JSP pages are used to dynamically generate HTML for consumption by a browser. A stand-alone client, such as the one described in the

administration scenario in section 10.1.1.2 on page 252, provides its own presentation. Control-related objects are present in each tier to enable coordination of actions across tiers. Objects that model business data and rules live in the EJB tier in an EJB-centric approach, and in the Web tier when using a Web-centric approach.

As discussed in several chapters in this book, the MVC architecture can be easily applied to enterprise applications. The presentation, business, and control categories map respectively, to the view, model, and controller concepts defined in the MVC architecture. The following sections take a detailed look at the design, implementation, and interactions of the sample application objects that constitute the view, model, and controller.

10.3 The View

The view determines the presentation of the user interface of an application. In the sample application, the implementation of the view is contained completely in the Web tier. In the sample application, three kinds of components work together to implement the view: JSP pages, custom JSP actions, and JavaBeans components.

JSP pages are used for dynamic generation of HTML responses. Custom JSP actions make it easier for JSP pages to use JavaBeans components when the underlying model is complex. Custom actions can also help encapsulate presentation logic and make it modular and more reusable.

JavaBeans components represent the contract between JSP pages and the model. JSP pages rely on these beans to read model data to be rendered to HTML, while elsewhere in the system, the model and controller coordinate to keep the JavaBeans components up to date.

This section describes the JSP pages and custom JSP actions that implement the view. Because the classes that implement JavaBeans components are intimately tied to their corresponding model classes, the discussion of the implementation of JavaBeans components and model classes is deferred until after the discussion of the model.

10.3.1 Shopping Interaction Interface

The shopping scenario described in section 10.1.1.1 on page 250 and the server-side scenario in section 10.2.1 on page 256 provide a behavioral specification for the shopping interaction interface. This section translates this specification into the set of views that the customer sees when interacting with the pet store application.

10.3.1.1 Screens

The user interface for the shopping interaction consists of a set of screens. A *screen* is the total content delivered to the browser when the user requests an application URL. In other words, a screen is what customers see when they navigate to one of our application's URLs. A screen can be composed of several components each contributing a different part of the its content.

The specification includes some notion of the kind of screens the application displays to the customer and the dialogs it carries on with them. Taking this process further, results in the specification of a complete set of screens, each with a unique name.The significance of the names will become clear later when the discussion turns to how the controller selects a view for each response. The following list identifies what model information it *presents* and what *user gestures* it can generate.

- Name: `MAIN_SCREEN, DEFAULT_SCREEN`

  This is the home page of the application. It displays a list of all product categories in the catalog, such as `Dogs`. The customer can click on any category to browse through a master view of products that belong in that category.

- Name: `CATEGORY_SCREEN`

  This screen displays a master view of all products that belong to a particular category. For each product it shows the product ID and its name. The customer can click on the name of any product on display to see further details of the product.

- Name: `SEARCH_SCREEN`

  This screen displays the results of a search. Searching the catalog is integral part of the application, and a search interface is displayed as part of every page. When the customer requests a search, the results are shown using the search screen. This screen is similar to the `CATEGORY_SCREEN` in that it displays a master view of the list of products that result from a search.

- Name: `PRODUCT_SCREEN`

  This screen displays information about a particular product. Each product can be offered for sale in several configurations. We call each of these an *item.* This screen lists inventory status for each item offered. The customer can

click on any item in inventory to see further details about it.

- Name: `PRODUCT_DETAILS_SCREEN`

  This screen displays detailed information about a particular product item, including a description of the item and its image and the number of items in the inventory. It also provides an Add button. Clicking this button adds the product currently being shown to the shopping cart and displays the resulting shopping cart.

- Name: `CART_SCREEN`

  This screen displays the contents of the customer's shopping cart. For each item in the shopping cart, it includes a brief description of the item and its quantity. The customer can change the quantity of each item and delete items from the cart. This screen includes an update button to update the cart according to the changes made by the customer. It also has a checkout button. Clicking the checkout button initiates the process of placing an order.

- Name: `CHECKOUT_SCREEN`

  This screen displays the final unmodifiable contents of the customer's shopping cart once again and asks the customer to confirm everything before placing the order. A customer confirms the order by clicking the place order button.

- Name: `PLACEORDER_SCREEN`

  This screen displays a form where the customer can fill in details necessary to place the order. A customer places the order by clicking the submit button.

- Name: `COMMIT_ORDER_SCREEN`

  This screen displays the receipt after an order has been confirmed and committed. It shows a unique order identifier so that the customer can track the order later on. It also shows a complete list of items ordered, the total price and shipping charges if any, as well as shipping and billing information.

- Name: `SIGNIN_SCREEN`

  This screen displays a customer name and password, allowing the customer to sign into the application. The submit button initiates the sign in pro-

cess.

- Name: `SIGNUP_SCREEN`

  This screen displays a form allowing a new customer to sign up and register themselves with the application. Once registered the customer can conveniently recall personal information each time they place an order.

This completes the initial set of screens that are presented to the customer during the course of a shopping interaction with the application. As the application evolves, more screens may be added and existing screens modified.

10.3.1.2 Graphical Design

Since we already identified the major screens in the application and the data that needs to be shown as part of each screen, we can now involve graphic design and HTML specialists to create the layout, look, and feel of each of the screens.

There are two parts to this design: design of the custom content of each screen and design of a common template which remains consistent with each of the screens. The preceding scenarios have identified the data that needs to go into each of the screens as well as the dynamic portions of the template.

The artist needs an idea of the size and shape of each of the data element that needs to be shown in each of the screens. They can make good progress with the graphical design at this point using storyboarding techniques, even without an actual implementation of the rest of the application available to them. This is one major advantage of decoupling the design of the user interface from the rest of the application.

The *contract* with the graphic design artists is just how the application makes the model *data* required for each screen available to the screen. As we shall see later in this section, this is where the JavaServer Pages technology comes into play.

10.3.2 JSP Pages

The JSP pages provided by the pet store application use a generic template mechanism and application-specific JavaBeans components. This section describes the template mechanism and discusses several example JSP pages. The JavaBeans components are discussed in section 10.5 on page 286. General guidelines on how to use JSP technology can be found in Chapter 4.

10.3.2.1 A Template Mechanism

While sketching out the shopping interaction interface of the application, it is clear that there are elements that we want to be part of each screen. Some of these are:

- The application logo and tag line.
- The search interface with a search text field and a search button.
- A help button to get information about the application.
- A show shopping cart button that provides immediate access to the shopping cart from any screen.
- A signin/signout status button that changes state based on whether the customer is signed in. If they are signed in, they are presented with a Sign out button. If they are not signed in, they see a Sign in button.
- Copyright notices and miscellaneous status information at the bottom of each page.

Among the elements that change on each page are the body and the title. Other elements, such as keywords and meta-headers, may also change with each screen as well.

To add headers and footers to every JSP page, the designer could create header and footer JSP files and have each JSP page include these at appropriate places. Such a technique is illustrated in Code Example 10.1.

```
<%@ include file="header.jsp" %>
   ...
   content of this screen
   ...
<%@ include file="footer.jsp" %>
```

Code Example 10.1 Templating Using JSP Include Statements

However, this approach runs into several limitations if we try to make the template more elaborate, using HTML tables, side bar, and so on. The header and footer files would have to be constructed and formatted properly to make sure the body appears where intended. For instance, correct HTML requires opening HTML tags in the header and a closing HTML tag in the footer so that the body

can be enclosed between them. This requires either hand-coded HTML or specialized authoring tools to ensure correct design and correct HTML. The problem becomes compounded if we want more than just one contiguous chunk of HTML to change on each screen. For instance, each screen might want to provide its own HTML title as well as custom content. These features require a more flexible screen layout mechanism.

A template mechanism provides a way to separate the common elements that are part of each screen from the elements that change with each screen. Putting all the common elements together into one file, makes it easier to maintain and enforce a consistent look and feel in all the screens. It also makes development of individual screens easier since the designer can focus on portions of a screen that are specific to that screen while the template takes care of the rest.

This section reviews the design and implementation of the sample application's screen template mechanism. The concept of a presentation template can be applied to almost any Web application in one form or another. The sample application's template mechanism is designed so that you can easily adapt it to other applications.

The template itself is a JSP page, with place holders for the parts that need to change with each screen. Each of these place holders is referred to as a *parameter* of the template. For example, a simple template could include a title text parameter for the top of the generated screen and a body parameter to refer to a JSP page for the custom content of the screen.

Once you have a template, you can generate different presentation screens from it simply by passing it different parameters. This process is called *instantiation* of the template. A specific screen is completely characterized by identifying the template page, and the parameters to pass to the template. The set of parameters that completely defines a screen is called a *screen definition.*While a large application could use multiple templates; the pet store application uses a single template for all its screens. However, the mechanism it uses is designed to support multiple templates.

From the templating mechanism's point of view a *screen* is the instantiation of a template according to its screen definition. Figure 10.3 illustrates the relationship between a template, a screen definition, and the resulting presentation screen.

Figure 10.3 Defining a Screen in Terms of a Template and its Parameters

Code Example 10.2 shows the contents of `template.jsp`, the template file used in the sample application.

```
<%@ page errorPage="errorpage.jsp" %>
<%@ page import="com.sun.estore.control.Web.ScreenNames" %>
<%@ taglib uri="WEB-INF/tlds/taglib.tld" prefix="j2ee" %>
<%@ include file="ScreenDefinitions.jsp" %>
<HTML >
    <head>
        <title>
            <j2ee:insert template="template"
                parameter="HTML Title" />
        </title>
    </head>
    <body bgcolor="white">
        <j2ee:insert template="template" parameter="HTML Banner" />
        <j2ee:insert template="template"
            parameter="HTML TopIndex" />
        <j2ee:insert template="template" parameter="HTML Body" />
    </body>
</HTML >
```

Code Example 10.2 `template.jsp`

The template is instantiated by forwarding to or dynamically including the `template.jsp` page. Forwarding is performed using the `jsp:forward` standard action or by calling the `forward` method of a `RequestDispatcher`; inclusion is per-

formed using an `include` directive or standard action or by calling the `include` method of a `RequestDispatcher`.

An appropriate screen definition must be set up in the request scope before invoking `template.jsp`. JSP pages can access objects in request, session, and application scopes. Since a screen is presented in the context of a specific URL request from the user, the appropriate screen definition needs to be set in the request scope before the template file is invoked. The other possible JSP scopes, session and application, are broader—they're more appropriate for setting general site and user-specific portions of the template.

The following examples show how `template.jsp` works. It uses the `j2ee:insert` custom tag to identify place holders in the template. This tag is responsible for extracting the screen definition from the request scope and inserting it into the page. Code Example 10.3 contains the implementation of the `insert` tag.

```
public class InsertTag extends TagSupport {
    private boolean directInclude = false;
    private String parameter = null;
    private String templateName = null;
    private Template template = null;
    private TemplateParameter templateParam = null;

    public void setTemplate(String templateName){
        this.templateName = templateName;
    }

    public void setParameter(String parameter){
        this.parameter = parameter;
    }

    public int doStartTag() {
        try {
            if (templateName != null){
                template = (Template)pageContext.getRequest().
                                getAttribute("template");
            }
        } catch (NullPointerException e){
            ...
        }
```

```
        if (parameter != null && template != null)
            templateParam = (TemplateParameter)template.
                getParam(parameter);
        if (templateParam != null)
            directInclude = templateParam.isDirect();
        return SKIP_BODY;
    }

    public int doEndTag() throws JspTagException {
        try {
            pageContext.getOut().flush();
        } catch (Exception e){
            ...
        }
        try {
            if (directInclude && templateParam != null) {
                pageContext.getOut().
                    println(templateParam.getValue());
            } else if (templateParam != null)  {
                if (templateParam.getValue() != null)
                pageContext.getRequest().
                    getRequestDispatcher(templateParam.
                        getValue()).include(pageContext.
                            getRequest(),
                                pageContext.getResponse());
            }
        } catch (Throwable ex) {
            ...
        }
        return EVAL_PAGE;
    }
}
```

Code Example 10.3 Insert Tag Implementation

The `insert` tag gets values of the parameters passed to it. The parameters are automatically set by the JSP runtime environment and the tag focuses on inserting

the appropriate parameters into the response. Parameters can be direct or indirect. Direct parameters are inserted as-is into the response stream. Indirect parameters are treated as the name of a JSP file, and that file is dynamically included into the response stream. This makes it possible to pass the title of a page as text using a direct parameter, and the body as the name of a JSP file to include using a direct parameter.

10.3.2.2 View Selection

In a Web application, each screen presented to the user can be considered as a different view. However, unlike the classic MVC architecture, all these views share the same controller. There needs to be a mechanism that allows the controller to choose a particular view to render in response to a user request. In the sample application, the controller makes this selection by specifying the screen ID of the screen to present as the response. This screen ID is mapped to a screen definition, then the template is instantiated.

Recall that the file `template.jsp` defines the template for the sample application. This file includes another file, `ScreenDefinitions.jsp`, which defines all the screens of the sample application. When the controller invokes the template file at request time, it sets the appropriate screen definition in the request scope. The template file passes this information to the screen definitions file which then returns the appropriate screen definition for the request.

One goal in structuring template and screen definition files is to facilitate internationalization (discussed in section 4.5 on page 102). This is achieved by separating text content from Java code. Since screen definitions that contain direct and indirect parameters are candidates for internationalization, we want to keep `ScreenDefinitions.jsp` devoid of Java technology code. We achieve this through the use of custom JSP actions. Code Example 10.4 contains an excerpt from `ScreenDefinitions.jsp`, which uses `Screen` and `Parameter` custom actions to pass text and the contents of files to the response.

```
<%@ page import="com.sun.estore.control.Web.ScreenNames" %>
<jsp:useBean
  id="screenManager"
  class="com.sun.estore.control.Web.ScreenFlowManager"
  scope="session"/>

<j2ee:CreateTemplate template="template"
    screen="<%=screenManager.getCurrentScreen(request)%>">
```

```
    <j2ee:Screen screen="<%=ScreenNames.MAIN_SCREEN%>">
        <j2ee:Parameter parameter="HTML Title"
            value="Welcome to Java Pet Store Demo" direct="true"/>
        <j2ee:Parameter parameter="HTML Banner"
            value="/banner.jsp" direct="false"/>
        <j2ee:Parameter parameter="HTML Body"
            value="/index.jsp" direct="false"/>
    </j2ee:Screen>
    <j2ee:Screen screen="<%=ScreenNames.SIGN_IN_SUCCESS_SCREEN%>">
        ...
    </j2ee:Screen>
    ...
</j2ee:CreateTemplate>
```

Code Example 10.4 ScreenDefinitions.jsp

When it is included at request time by the template file, `ScreenDefinitions.jsp` uses `ScreenFlowManager`, a component of the controller, to identify the view that the controller wishes to select. The nested custom tags arrange for that screen's definition to be set into the request scope when this file invoked.

In summary, the JSP pages `template.jsp` and `ScreenDefinitions.jsp` work together to create the page viewed by the user. Figure 10.4 depicts the process of view selection and instantiation in the sample application.

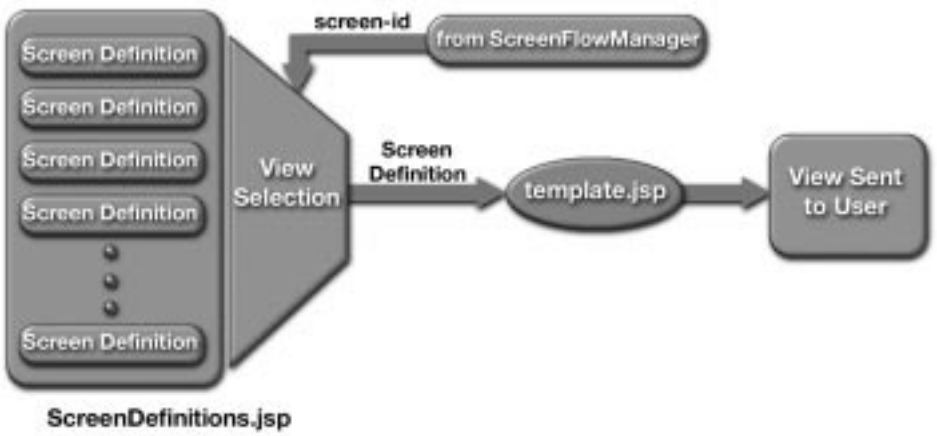

Figure 10.4 View Selection and Instantiation

10.3.3 Examples

For the most part, the sample application's presentation JSP pages use fairly straightforward JSP elements. This section examines example three presentation JSP pages: the home screen page (`index.jsp`), the products-by-category page (`productcategory.jsp`), and the shopping cart page (`cart.jsp`).

10.3.3.1 index.jsp

The home screen of the Java Pet Store Demo application is shown in Figure 10.5.

Figure 10.5 Home Screen

The JSP source used to generate the screen, contained in the file `index.jsp`, appears in Code Example 10.5. The screen is composed of the banner that appears in all the Java Pet Store Demo screens, a list of the product categories (`sideindex.jsp`) supported by the application, and an imagemap (`splash.jsp`) of the categories. The banner does not appear explicitly, because is constructed by the template described in section 10.3.2.1 on page 267.

```
<table border="0"  cellspacing="0" width="600" >
  <tr>
    <td>
      <%@ include file="sideindex.jsp"%>
    </td>
    <td bgcolor="white" height="300">
```

```
        <%@ include file="splash.jsp" %>
      </td>
    </tr>
  </table>
  </HTML >
```

Code Example 10.5 index.jsp

10.3.3.2 productcategory.jsp

The screen that lists all the products in a category is shown in Figure 10.6.

Figure 10.6 Product Category Screen

The JSP source used to generate the screen, contained in the file `productcategory.jsp`, appears in Code Example 10.6. In this code sample, the first statement sets the `catalog` variable to point to an instance of the JavaBeans component `CatalogWebImpl`. This component is used to retrieve the catalog entries for a particular product category. The category is retrieved from the implicit `request` object with the `getParameter("category_id")` method. Once the category and its products are retrieved, JSP scriptlets are used to generate the table of products.

```
<jsp:useBean
  id="catalog"
  type="com.sun.estore.catalog.model.CatalogModel"
  class="com.sun.estore.catalog.web.CatalogWebImpl"
```

```
  scope="application"/>

<%
String key = request.getParameter("category_id");
Category category = null;
if (key != null)  category = catalog.getCategory(key);
if (category != null) {
    Collection products = null;
    products = catalog.getProducts(key, 0, 20);
%>
<p>
<font size="5" color="green"><%= category.getDescription()%></font>
<p>

<table border="0" bgcolor="#336666">
<tr background="../images/bkg-topbar.gif">
    <th><font color="white" size="3">Category ID</font></th>
    <th><font color="white" size="3">Category Name</font></th>
</tr>

<%
    Iterator it = null;
    if (products != null) it =products.iterator();
    while (it.hasNext()) {
        Product product = (Product)it.next();
%>
    <tr bgcolor="#eeebcc">
        <td><%= product.getId() %></td>
        <td>
        <a href="product?product_id=<%= product.getId() %>">
         <%= product.getName()%>
      </a>
        </td>
    </tr>

<% } %>

</table>
<%
```

```
    } else {
   // Category was not found:
%>
<p>
<font size="5" color="red">Unable to Locate Category ID <%= key
%></font>

<% } %>
```

Code Example 10.6 productcategory.jsp

10.3.3.3 cart.jsp

The screen that displays the contents of a user's shopping cart is shown in Figure 10.7.

Figure 10.7 Shopping Cart Screen

The JSP source used to generate the screen, contained in the file `cart.jsp`, appears in Code Example 10.7 and Code Example 10.8. In this code sample, the first statement sets the `cart` variable to point to an instance of the JavaBeans component `ShoppingCartWebImpl`. This component is used by the shopping cart table. The include statement towards the middle of the code sample (begins with "`<%@ include`") causes the page to include the shopping cart table (illustrated in Code Example 10.8). The page also contains a button that accepts a modification to the shopping cart, and a link to a checkout page.

```
<jsp:useBean
    id="cart"
    class="com.sun.estore.cart.Web.ShoppingCartWebImpl"
    scope="session"
<%
    cart.init(session);
%>
</jsp:useBean>
<p>
<%
  if (cart.getSize() > 0) {
%>

<font size="5" color="black">Shopping Cart:</font>
<p>
  <form action="cart">
  <input type="hidden" name="action" value="updateCart">
  <table bgcolor="white">
    <tr>
      <td>
        <%@ include file="changeable_carttable.jsp" %>
      </td>
      <td>
        <input type="image" border="0" src="../images/cart-up-
date.gif" name="update">
      </td>
    </tr>
  </table>
  </form>
  <a href="checkout"><img src="../images/button_checkout.gif"
alt="Proceed To Checkout" border="0"></a>

<%
  } else {
  // The cart is empty
%>
```

```
  <font size="5" color="red">Shopping Cart is empty.</font>
<% } %>
```

Code Example 10.7 cart.jsp

Code Example 10.8 uses JSP scripting capabilities to display all the rows in the shopping cart. The page retrieves shopping cart items from the `cart` component set by the enclosing page `cart.jsp`. The page also includes a button that allows a user to delete an item from the shopping cart.

```
<table bgcolor="#336666">

    <tr background="../images/bkg-topbar.gif" border="0">
     <th><!-- for the remove column --></th>
    <th><font size="3" color="white">Item ID</font></th>
    <th><font size="3" color="white">Product Name</font></th>
    <th><font size="3" color="white">In Stock</font></th>
    <th><font size="3" color="white">Unit Price</font></th>
    <th><font size="3" color="white">Quantity</font></th>
    <th><font size="3" color="white">Total Cost</font></th>
    </tr>

<%--
% Loop through each item in the shopping cart.  The current item is
% available to the jsp block within the loop as "item"
--%>

<%
  Iterator it = cart.getItems();
  while ((it != null) && it.hasNext()) {
    CartItem item = (CartItem)it.next();
%>

<tr bgcolor="#eeebcc">
<td>
    <a href="cart?action=removeItem&itemId=<%=item.get-
ItemId()%>"><img src="../images/button_remove.gif" border="0"
alt="Remove Item From Shopping Cart">
```

```
    </a>
</td>

<td><%= item.getItemId() %></td>
<td>
    <a href="productdetails?item_id=<%=item.getItemId()%>">
    <%=item.getAttribute()%> <%=item.getName()%>
    </a>
</td>

<td><%/* item.isInStock() */%></td>
<td><%=JSPUtil.formatCurrency(item.getUnitCost())%></td>
<td><input name="itemQuantity_<%=item.getItemId()%>"
    type="text"
    size="4"
    value="<%=item.getQuantity()%>">
</td>
<td><%=JSPUtil.formatCurrency(item.getTotalCost())%></td>
</tr>

<%
  } // end for loop
%>

<tr background="../images/bkg-topbar.gif">
    <td></td>
    <td><font size="3" color="white">Total:</font></td>
    <td></td>
    <td></td>
    <td></td>
    <td></td>
    <td><font size="3" color="white"> <%=JSPUtil.formatCurren-
cy(cart.getTotalCost())%> </font></td>

</tr>
</table>
```

Code Example 10.8 changeable_carttable.jsp

10.4 The Model

In this section we focus our attention on the state that needs to be maintained by the application. One can think of the back-end of an application as a collection of state with some rules on how the state changes in response to user interactions. This section explains how the sample application maintains state in the J2EE platform and persistent data in database tables.

10.4.1 State in the J2EE Platform

Typically, the customer will use a number of features of the pet store during a single visit (such as requesting product information and placing items in a shopping cart), resulting in numerous requests during the client session. While the application does not need to store this information in a database, this information must be somehow tracked to maintain a meaningful dialog between the customer and the application.

There is state associated with both the user interface and the business logic. In general, the sample application must maintain the following state:

- The user identity - Typically, the user account module maintains the user identify, which includes the user's login ID and certain security credentials.
- The search cursor and catalog position - The catalog module maintains the cursor's position within the current search and within the catalog hierarchy.
- The items in the shopping cart - The shopping cart module maintains the list of items placed in the customer's shopping cart.
- Order information - When the customer confirms the order, the shopping cart passes this information the order information—billing address, shipping address, and payment method—to the order management module, which eventually stores it to a database.

The J2EE platform provides several choices for storing the application state. An application can store state in the Web tier using the state maintenance capabilities of servlets, which include the `HTTPSession` and `ServletContext` objects as well as JavaBeans components. In the EJB tier, state can be maintained using enterprise beans. Also, session state for an application can be divided between these tiers. The decision of where each object representing application state is stored depends on the lifetime and scope of the object. The following sections

identify each state component, its lifetime requirements, and discuss why it should be stored using a particular mechanism.

The Web tier maintains state required by JSP pages in JavaBeans components. These JavaBeans components are managed by a class called `ModelManager` that uses both an `HttpSession` and a `ServletContext` to maintain handles to the JavaBeans components. `ModelManager` is discussed further in section 10.6.8.1 on page 302. Beans that are specific to a client are maintained by an HTTP session object. Beans that can be shared by all clients are maintained as an attribute of the servlet generated from `Main.jsp`.

The JavaBeans components contain copies of the state maintained by corresponding model objects which are maintained in the EJB tier. When designing objects in the EJB tier to maintain state, the developer must answer two questions:

- What is the appropriate granularity for the objects? Not every business object should be modelled as an enterprise bean. Since every method call to an enterprise bean is potentially a remote call, the overhead of an inter-component call is likely to be prohibitive for interactions with fine-grained objects. Therefore, the sample application makes extensive use of helper objects, which are non-remote, serializable objects that mirror their respective enterprise beans.
- What type of enterprise bean should I use? An application can use either session beans or entity beans to maintain state. For a non-transactional object, a session bean is the simplest way to maintain session state for a short period of time because it leverages the EJB container's ability to manage session bean state. Using entity beans to maintain state provides transactional support for storing the state data in the database. While there is overhead in making the object transactional, the object reference could persist for as long as needed, even beyond the scope of a single session. For example, an object reference can be stored in a cookie on the browser to be retrieved and used even weeks later. The sample application has examples of using both session and entity beans to store session state.

10.4.1.1 Using Enterprise Beans to Maintain Session State

This section describes how different types of enterprise beans are used to represent objects in the sample application. General guidelines for how to use enterprise beans can be found in Chapter 5.

10.4.1.1.1 Stateless Session Beans

A stateless session bean does not contain state for a specific client. However, the instance variables of a stateless session bean can contain state across method calls. Examples of such state include an open database connection and a cache of data retrieved from that connection. Stateless session beans are never written out to secondary storage. As a consequence, stateless beans usually offer better performance than stateful beans.

The sample application uses stateless session beans for objects containing more than one database row. In particular, because stateless session beans provide high performance, stateless session beans are a good choice to provide a fast access to data derived from multiple database rows. In the pet store application, the `Catalog` stateless session bean functions as a cache that is built up over time.

10.4.1.1.2 Stateful Session Beans

A stateful session bean exists during a single client session and can maintain information specific to a client between invocations of methods. The sample application represents the contents of a client's shopping cart with the `ShoppingCart` stateful session bean.

10.4.1.1.3 Entity Beans

The sample application uses entity beans to provide an object view of individual rows in a database. The sample application includes three such beans, `Account`, `Inventory`, and `Order`, to represent individual rows in the corresponding tables.

10.4.1.2 Helper Objects

It is not appropriate to model all objects in the EJB tier as enterprise beans. Therefore, the sample application uses helper objects that are subordinate to their respective enterprise beans for a number of purposes. The different types of helper objects are: data access objects and value objects. The use of helper objects is discussed in detail in section 5.5 on page 142.

10.4.1.2.1 Data Access Objects

A data access object is used to encapsulate access to databases. Data access objects can encapsulate access to more than one database, more than one table within one database, and different types of databases. The sample application uses data access objects for all these purposes.

The sample application uses the abstract data access class `OrderDAO` to access three tables, `order`, `orderstatus`, and `lineitem`, when an order is created, read, or updated. The sample application contains three subclasses, `OrderDAOOracle`, `OrderDAOSybase`, and `OrderDAOCS`, that are used to access Oracle, Sybase, and Cloudscape databases.

10.4.1.2.2 Value Objects

A value object is a business object that can be passed by value as a serializable Java object. A business concept should be implemented as a value object when it is fine-grained, dependent, and immutable. The sample application uses two types of value objects: dependent objects and details objects.

An object is a dependent object of another object if its life cycle is completely managed by that object and if it can only be accessed indirectly through that object. Examples of dependent objects in the sample application are `Address` and `CreditCard`.

A value object can also be used to encapsulate an entire remote object. Such objects allow a client to retrieve the value of a remote object in one remote call. The sample application contains details objects for each enterprise bean. Code Example 10.9 illustrates an account entity bean and its corresponding details object. In keeping with its purpose, `AccountModel`'s methods only enable retrieval of the values in the fields of its bean, while `Account` itself provides a method for setting a value and a coarse-grained method (`getDetails`) that returns an `AccountModel`.

```
public interface Account extends EJBObject {
    public AccountModel getDetails() throws RemoteException;
    public void changeContactInformation(ContactInformation info)
        throws RemoteException;
}

public class AccountModel implements java.io.Serializable {
    private String userId;
    private String status;
    private ContactInformation info;
    ...
    public String getUserId() {
        return userId;
    }
```

```
    public ContactInformation getContactInformation() {
        return info;
    }
    ...
}
```

Code Example 10.9 Account and AccountModel

10.4.2 Persistent Data

The sample application maintains persistent data in database tables, organized according to the functional areas of the application. Figure 10.8 illustrates the database schema

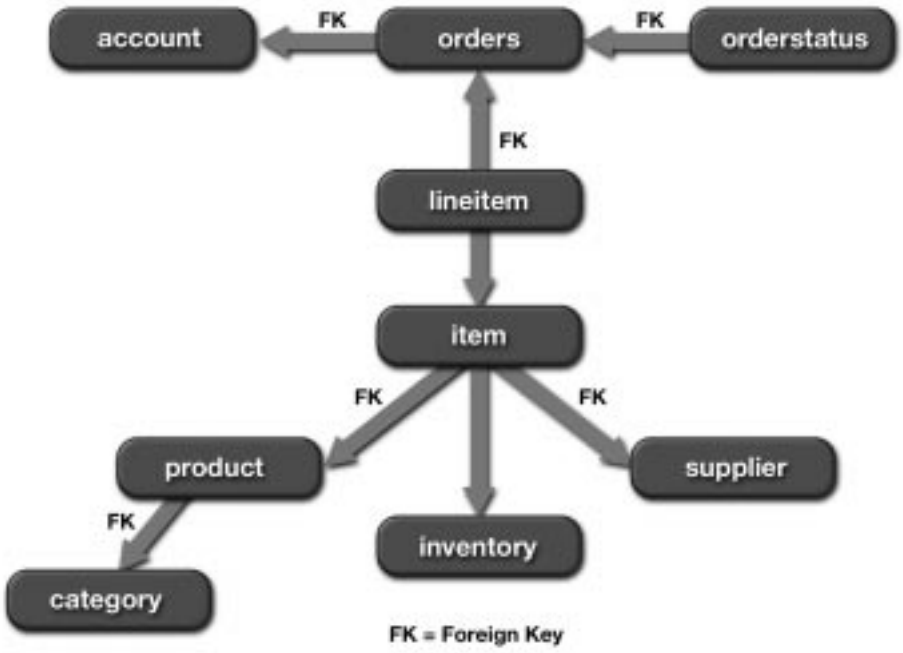

Figure 10.8 Database Tables and Relationships

The application uses this database schema to maintains accounts and tracks orders for products. Thus, there are three areas for which data must be maintained: product, account, and order information. The `product`, `category`, and `item` tables

represent the business's product catalog. Each item has an associated entry in the `inventory` table that represents the inventory for that product. The `account` table maintains customer account information, one record per customer, with information such as customer name, password, and customer address. Finally, there is an `orders` table with one record per order, for information such as ship-to address, bill-to address, total price of the order, and payment (credit card name, expiration date, type) information. The `orders` table is linked to `lineitem` and `orderstatus` tables. Each item in an order is stored in a separate `lineitem` record, which contains the quantity ordered and price and a separate `orderstatus` record, which contains a reference to the item and the status of the order.

10.5 Implementation

In the implementation of the view, JSP pages rely on JavaBeans components to mirror model data. These components are named *ESObject*`WebImpl` (where *ESObject* are the e-store objects `Inventory`, `Account`, `Cart`, and `Order`). As Code Example 10.10 illustrates, *ESObject*`WebImpl` extends *ESObject*`Model` and implements a listener interface so that views can be notified of changes to their corresponding models. For example, when an `AccountWebImpl` is created, it adds itself to the list of listeners interested in updates to the account model. When an account model changes, the manager of the view objects invokes the `performUpdate` method on all views that have registered as listeners of the account model. See section 10.6.8 on page 301 for further discussion of model-view synchronization.

```
public class AccountWebImpl extends AccountModel
    implements ModelUpdateListener {

    private ModelManager mm;
    private Account acctEjb;

    public AccountWebImpl(ModelManager mm) {
        super(null, null, null);
        this.mm = mm;
        mm.addListener(JNDINames.ACCOUNT_EJBHOME, this);
    }

    public void performUpdate() {
    // Get data from the EJB
```

```
            if (acctEjb == null) {
                acctEjb = mm.getAccountEJB();
            }
            try {
                if (acctEjb != null) copy(acctEjb.getDetails());
            } catch (RemoteException re) {
                throw new GeneralFailureException(re);
            }
        }
    }
```

Code Example 10.10 AccountWebImpl

The model is implemented by enterprise beans named *ESObject*. These beans are supported by data access classes named *ESObject*DAO and details classes named *ESObject*Model. As described in section 10.4.1.2.2 on page 284, a client can retrieve the contents of an enterprise bean with one remote call that returns a details object.

JavaBeans components and details classes share aspects of their implementation (that is, the *ESObject*Model classes), because the *ESObject*Model classes capture the essential information required to represent e-store business objects in any tier.

The implementation of the catalog does not follow the pattern just described because it implemented in both a Web-centric and EJB-centric fashion. The Web-centric design is used for high performance since the catalog is read-only and the most frequently accessed object in the system. Thus the Web-centric JavaBeans component CatalogWebImpl accesses the data access class CatalogDAO directly instead of calling an enterprise bean.

Since the shopping cart enterprise bean needs access to the catalog and cannot access the Web-tier catalog it uses a catalog enterprise bean. Note that high performance is not as crucial in this case as compared to the earlier case but access to the catalog is still read-only.

The implementation of the catalog functionality is essentially the same in both cases, so both CatalogWebImpl and CatalogEJB extend CatalogImpl which implements the CatalogModel interface.

The relationships between the sample application business objects—view classes in the Web tier, model classes (and their respective helper classes) in the

EJB tier, and database tables in the enterprise information system tier—are shown in Figure 10.9.

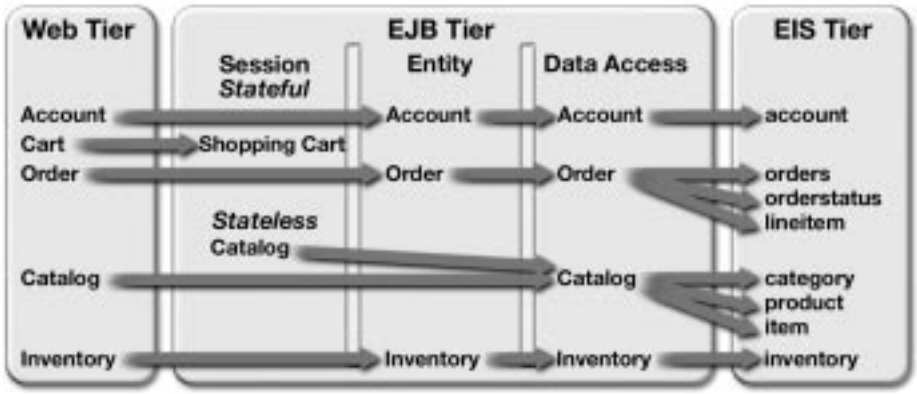

Figure 10.9 Sample Application Business Objects

10.6 The Controller

The sample application must reflect the state of a user's interaction with the application and the current values of persistent data in the user interface. Following the MVC architecture, this functionality is implemented within the controller. In the sample application, the controller is split between the Web tier and the EJB tier. In this section we will discuss the implementation of the controller for the shopping interaction in the sample application.

The controller is responsible for coordinating the model and view. As described in section 10.2 on page 256, the view depends on the controller for screen selection. The model depends on the controller for making state changes to the model. The controller must accept user gestures from the view, translate them into business events based on the behavior of the application, and process these events. The processing of an event involves invoking methods of the model to cause the desired state changes. Finally, the controller selects the screen shown in response to the request that was processed.

Since the controller must coordinate both the view and the data, it straddles the Web and EJB tiers. Some components of the controller are hosted by the Web tier and facilitate communication with the view, while others are hosted by the EJB tier and control the model.

In the Web tier, the controller consists of several components:

- `Main.jsp` receives and processes HTTP requests. `Main.jsp` calls `ScreenFlowManager`, which is responsible for selecting the next screen to be shown to the client after the completion of the current request.
- `RequestProcessor` provides the glue in the Web tier for holding the application components together. It contains logic that needs to be executed for each request. `RequestProcessor` collaborates with two classes:
  - `RequestToEventProcessor` translates HTTP requests into business event that the rest of the application can operate on. Events are represented by the class `eStoreEvent` and its subclasses `CatalogEvent`, `LoginEvent`, `AccountEvent`, `CartEvent`, and `OrderEvent`.
  - `ShoppingClientControllerWebImpl` (`SCCWI`) provides a Web-tier proxy for `ShoppingClientController`. It delegates all methods to its EJB tier counterpart.

In the EJB tier, the controller is `ShoppingClientController` (`SCC`), which provides the view with read-only access to the model and handles business events. `ShoppingClientController` collaborates with `StateMachine`, an object that controls the creation and removal of enterprise beans and handles events to modify those objects passed to it by the controller in the Web tier.

Figure 10.10 illustrates the interactions that occur between the collaborating controller objects when an HTTP request is handled. The servlet generated from `Main.jsp` receives all HTTP requests. It passes the request to `RequestProcessor`, which coordinates all handling of the request. `RequestProcessor` uses `RequestToEventTranslator` to translate the HTTP request into a business event. `RequestProcessor` then passes the event to `ShoppingClientControllerWebImpl`, which forwards the event to `ShoppingClientController`, the controller in the EJB tier. `ShoppingClientController` delegates the handling of the business event to `StateMachine`. `StateMachine` changes the state of the model in response to the business event or command and then retrieves a list of model objects that have changed

as a result of handling the business event from `ModelUpdateManager` (`MUM`). Finally, `RequestProcessor` notifies all registered views of model changes.

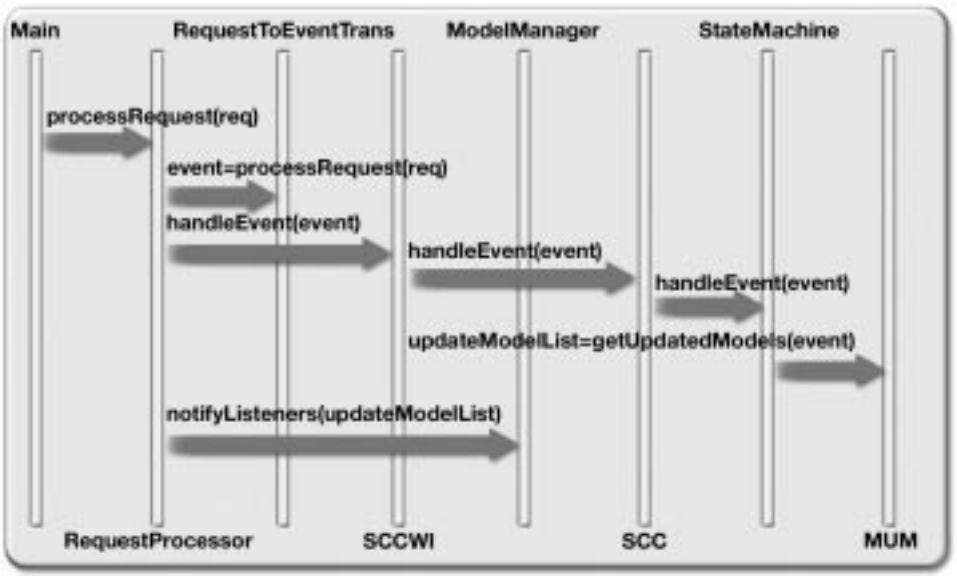

Figure 10.10 Controller Object Interaction Diagram (Part 1)

Figure 10.11 shows what happens after `RequestProcessor.processRequest` returns. `Main.jsp` forwards the initial request to `template.jsp`. The template includes `ScreenDefinitions.jsp`, which uses `ScreenFlowManager` to map the screen to a JSP page.

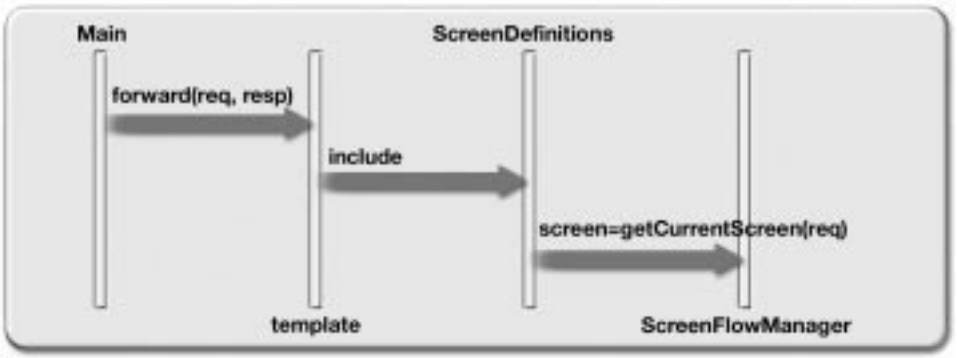

Figure 10.11 Controller Object Interaction Diagram (Part 2)

In the following sections, we will discuss the implementation of each of these components in more detail.

10.6.1 Main

A front component is a component to which all requests for application URLs are delivered. The front component `Main.jsp`, processes these requests and delegates the generation of the response to the template page.

Code Example 10.11 shows `Main.jsp`. The highlighted lines in the example indicate these two steps. `Main.jsp` delegates all of the request processing tasks to `RequestProcessor`. The response is generated by forwarding to `template.jsp`. An interesting detail to note here is that `Main.jsp` stores references to the request processor and other session-specific beans in the HTTP session object.

```
<jsp:useBean id="modelManager"
    class="com.sun.estore.control.Web.ModelManager" scope="ses-
sion">
    <% modelManager.init(config.getServletContext(), session); %>
</jsp:useBean>
<jsp:useBean id="rp"
    class="com.sun.estore.control.Web.RequestProcessor"
    scope="session">
    <% rp.init(config.getServletContext(), session); %>
</jsp:useBean>
<%
try {
    rp.processRequest(request);
    request.setAttribute("selectedURL" , request.getPathInfo());
    } catch (MissingFormDataException mi){
        request.setAttribute("missingformdata", mi);
        request.setAttribute("selectedURL", "/missingformdata");
    } catch (DuplicateAccountException du){
        request.setAttribute("selectedURL", "/duplicateaccount");
}
    getServletConfig()
        .getServletContext()
        .getRequestDispatcher("/template.jsp")
```

```
            .forward(request, response);
    %>
```

Code Example 10.11 Main.jsp

10.6.2 RequestProcessor

`RequestProcessor` contains logic that gets executed for each request. For example, when a customer tries to access a feature that requires signin, `RequestProcessor` checks to detect whether the customer is logged in.

Code Example 10.12 presents an excerpt from `RequestProcessor`, simplified to illustrate the key aspects of its behavior.

```
public class RequestProcessor {
    private ShoppingClientControllerWebImpl scc;
    private ModelManager mm;
    private ModelUpdateNotifier mun;
    private RequestToEventTranslator eventTranslator;
    private SecurityAdapter securityAdapter;
    public void init(...) {
        mm = (ModelManager)session.getAttribute("modelManager");
        mun = mm;
        scc = new ShoppingClientControllerWebImpl(session);
        eventTranslator =
            new RequestToEventTranslator(this, mm);
        ...
     }
     public void processRequest(HttpServletRequest req) {
        checkForWebServerLogin(req);
        EStoreEvent event = eventTranslator.processRequest(req);
        if (event != null) {
            Collection updatedModelList = scc.handleEvent(event);
            mun.notifyListeners(updatedModelList);
        }
        ...
    }
}
```

Code Example 10.12 `RequestProcessor`

This excerpt demonstrates the core responsibilities of `RequestProcessor` including:

- Initializing the client session. `RequestProcessor` instantiates an object that implements `ShoppingClientController` and related application objects when a new session is initiated.
- Detecting when the user logs into the server using form-based authentication and generating a login business event when this happens.
- Computing the business event to generate based on the `HttpRequest` that came in, with the help of the `RequestToEventTranslator`.
- Raising a business event by invoking `handleEvent` on the `ShoppingClientController`'s Web implementation.
- Gathering the outcome of the event processing. In particular, `RequestProcessor` passes the business event and its outcome to the `ModelManager` so the model change notifications can be processed by the view components (see section 10.6.8 on page 301).

10.6.3 RequestToEventTranslator

`RequestToEventTranslator` is responsible for taking an HTTP-specific request and converting it into a business event that is not tied to the specifics of the HTTP protocol.

Application objects that include HTTP-specific functionality are not easy to reuse. By removing HTTP protocol-specific details from the request as early as possible, by turning it into a business event, the sample application ensures that all components that deal with business events would be completely reusable with non-HTTP clients. For example, the `StateMachine` that implements command processing logic for the sample application could be easily used as-is by a standalone Java client.

The two standard HTTP requests that can be processed by the translator are `GET` and `POST`. It is relatively straightforward to map `GET` requests to business events. However, `POST` requests, which represent form submission in the sample application, require the request processor to validate the form data as part of generating the business event. The processor needs to keep track of the values entered in the form so that the presentation screen can show where the error occurred

when the form data is invalid. When the form data is valid, the processor must encapsulate the form parameters in an application-specific business event.

Code Example 10.13 presents excerpts from `RequestToEventTranslator`. The highlighted the lines indicate where the translator parses HTTP request parameters and converts them to objects to be used in business events.

```
public class RequestToEventTranslator {
    private ModelManager mm;
    public EStoreEvent processRequest(HttpServletRequest req)
        throws EStoreEventException,  MissingFormDataException {
        String selectedUrl = req.getPathInfo();
        EStoreEvent event = null;
        if (selectedUrl.equals(ScreenNames.CATALOG_URL)) {
            event = createCatalogEvent(req);
        else if (selectedUrl.equals(ScreenNames.CART_URL)) {
            mm.getCartModel();
            event = createCartEvent(req);
        } else if ...
        return event;
    }
    private EStoreEvent createCatalogEvent(HttpServletRequest req) {
        CatalogEvent event = null;
        String[] category = req.getParameterValues(CATEGORY_ID );
        if (category != null) {
            event = new CatalogEvent(
                CatalogEvent.BROWSING_EVENT, category[0]);
        }
        return event;
    }
    private CartEvent createCartEvent(HttpServletRequest request) {
        String action = request.getParameter("action");
        if (action.equals("purchaseItem")) {
            return createPurchaseItemEvent(request);
        } else if (action.equals("removeItem")) {
            return createRemoveItemEvent(request);
        } else if (action.equals("updateCart")) {
            return createUpdateCartEvent(request);
        }
```

```
        }
    }
```

Code Example 10.13 RequestToEventTranslator

10.6.4 ShoppingClientControllerWebImpl

`ShoppingClientControllerWebImpl` is a proxy object that calls methods on the EJB tier controller `ShoppingClientController`. `ShoppingClientControllerWebImpl` exposes a read-only interface to the model, so that the view can render the model as needed. Keeping this interface read-only minimizes dependencies between the view and the model, to prevent inadvertent modification of the model by the view outside the scope of the business rules encapsulated in the application.

Code Example 10.14 contains an excerpt from `ShoppingClientControllerWebImpl`. Notice that all the methods of `ShoppingClientController` are synchronized so that concurrent requests to `ShoppingClientController` are serialized. This is done because an EJB container will throw an exception if a request is made to a session bean while it is servicing another request.

```
public class ShoppingClientControllerWebImpl
{
    private com....ejb.ShoppingClientController sccEjb;
    private HttpSession session;
    public ShoppingClientControllerWebImpl(HttpSession session) {
        this.session = session;
        ModelManager mm =
            (ModelManager)session.getAttribute("modelManager");
        sccEjb = mm.getSCCEJB();
    }
    public synchronized AccountModel getAccount() {
        return sccEjb.getAccount().getDetails();
    }
    ...
    public synchronized Collection handleEvent(EStoreEvent ese) {
        return sccEjb.handleEvent(ese);
    }
    public synchronized void remove() {
        sccEjb.remove();
```

```
        }
    }
```

Code Example 10.14 ShoppingClientControllerWebImpl

10.6.5 ShoppingClientController

`ShoppingClientController` manages the life cycle of model objects such as the shopping cart and account enterprise beans and processes business events. It delegates the processing of business events in the `handleEvent` method to `StateMachine`. `ShoppingClientController` is also responsible for the life cycle of `StateMachine`. `ShoppingClientController` is implemented by `ShoppingClientControllerEJB`, illustrated in Code Example 10.15.

```
public class ShoppingClientControllerEJB implements SessionBean {
    private StateMachine sm;
    private ShoppingCart cart;
    String userId;
    Account acct;

    public Account getAccount() {
        if (acct == null) {
            createAccountEJB();
        }
        return acct;
    }
    public ShoppingCart getShoppingCart() {
        if (cart == null) {
            try {
                ShoppingCartHome cartHome =
                    EJBUtil.getShoppingCartHome();
                cart = cartHome.create();
            } catch (CreateException ce) {
                throw new EJBException(ce);
            }
        }
        return cart;
    }
    public void ejbCreate() {
```

```
            sm = new StateMachine(this);
        }
        public Collection getOrders() throws FinderException {
            Collection orders = null;
            if (userId != null) {
                OrderHome home = EJBUtil.getOrderHome();
                orders = home.findUserOrders(userId);
            }
            return orders;
        }
        public Collection handleEvent(EStoreEvent ese)
            throws EStoreEventException {
            try {
                return (sm.handleEvent(ese));
            } catch (RemoteException re) {
                throw new EJBException (re);
            }
        }
    }
```

Code Example 10.15 ShoppingClientControllerEJB

10.6.6 StateMachine

`StateMachine` implements the core command processing business logic of the application. It is responsible for changing the state of the models in response to a business event or command. `StateMachine` consists of methods that handle each of the different business events that the sample application can respond to. One such method is highlighted in Code Example 10.16.

```
    public class StateMachine {
        private ShoppingClientControllerEJB sccejb;
        private ModelUpdateManager mum;
        private HashMap orderTable;
        public StateMachine(ShoppingClientControllerEJB sccejb) {
            this.sccejb = sccejb;
            this.mum = new ModelUpdateManager();
        }
        public Collection handleEvent(EStoreEvent ese)
```

```
            throws RemoteException, EStoreEventException {
            if (ese instanceof CartEvent) {
                handleCartEvent((CartEvent)ese);
            } else if (ese instanceof AccountEvent) {
                handleAccountEvent((AccountEvent)ese);
            } else if (ese instanceof OrderEvent) {
                handleOrderEvent((OrderEvent)ese);
            } else if (ese instanceof LoginEvent) {
                login((LoginEvent)ese);
            } else if (ese instanceof LogoutEvent) {
                logout();
            }
            return (mum.getUpdatedModels(ese));
        }
        private void handleCartEvent(CartEvent ce)
            throws RemoteException {
            ShoppingCart cart = sccejb.getShoppingCart();
            switch (ce.getActionType()) {
                ...
                case CartEvent.UPDATE_ITEM :{
                    Collection itemIds = ce.getItemIds();
                    Iterator it = itemIds.iterator();
                    while (it.hasNext()){
                        String itemId = (String)it.next();
                        int quantity = ce.getItemQty(itemId);
                        if (quantity > 0){
                            cart.updateItemQty(itemId, quantity);
                        } else {
                            cart.deleteItem(itemId);
                        }
                    }
                }
                break;
            }
        }
        ...
    }
```

Code Example 10.16 StateMachine

StateMachine has both read and write access to all of the model objects so that it can coordinate event processing across multiple model objects. For example, when StateMachine handles an order event, it interacts with the inventory bean to debit the quantity of the purchased item, the order bean to insert the order details, and the mailer bean to send confirmation email to the user. These functions are performed by the method illustrated in Code Example 10.17. Highlighted lines indicate where enterprise beans are retrieved or created.

```
private Order createOrder(OrderEvent oe) throws RemoteException {
    ShoppingCart cart = sccejb.getShoppingCart();
    Order order = null;
    String userId = sccejb.getAccount().getDetails().getUserId();
    try {
        InventoryHome inventHome = EJBUtil.getInventoryHome();
        Iterator ci = ((ShoppingCartModel)cart.getDetails()).
            getItems();
        ArrayList lineItems = new ArrayList();
        int lineNo = 0;
        double total = 0;
        while (ci.hasNext()) {
            lineNo++;
            CartItem cartItem = (CartItem) ci.next();
            LineItem li = new LineItem(cartItem.getItemId(),
                cartItem.getQuantity(),cartItem.getUnitCost(),
                lineNo);
            lineItems.add(li);
            total += cartItem.getUnitCost() * cartItem.getQuantity();
        }

        for (Iterator it = lineItems.iterator(); it.hasNext();){
            LineItem LI = (LineItem)it.next();
            Inventory inventRef =
                inventHome.findByPrimaryKey(LI.getItemNo());
            inventRef.updateQuantity(LI.getQty());
        }

        OrderHome home = EJBUtil.getOrderHome();
            order = home.create(lineItems,
                oe.getShippingAddress(),
```

```
                oe.getBillingAddress(),
                ...
                total);

    // put the requestId and the orderId in a table to match up later
            if (orderTable == null) orderTable = new HashMap();
            orderTable.put(oe.getRequestId() + "",
                order.getDetails().getOrderId() +"");

            // empty shopping cart
            cart.empty();

            if (JNDIUtil.sendConfirmationMail()) {
                // send order confirmation mail.
                Mailer mailer = EJBUtil.createMailerEJB();
                mailer.sendOrderConfirmationMail(order.getDetails().
                    getOrderId());
            }
        } catch (DuplicateKeyException dke) {
            ...
        } catch (CreateException ce) {
            throw new EJBException(ce);
        } catch (FinderException fe) {
            throw new EJBException(fe);
        }
        return order;
    }
```

Code Example 10.17 StateMachine.createOrder

10.6.7 ScreenFlowManager

`ScreenFlowManager` is responsible for selecting a screen to present to the user as the outcome of their request. The mapping from a requested URL to a response screen is not one to one. In fact, the response that depends not only on the request itself, but also on the state of the application data model and the outcome of request processing within the application. In other words, the flow manager keeps a state machine that captures the flow of screens in the application. `ScreenFlowManager` looks at the request and the state of the model and computes the screen to be returned.

Code Example 10.18 shows how ScreenFlowManager maps many of the request URLs directly into response screens. For some of the requests, such as the VALIDATE_BILLING_INFO_URL, the code inspects the model to decide which of two possible screens to present.

```
public class ScreenFlowManager {
    public int getCurrentScreen(HttpServletRequest req)  {
        String selectedUrl =
            (String)req.getAttribute("selectedURL");
        int nextScreen = ScreenNames.DEFAULT_SCREEN;
        if (selectedUrl == null) {
            // do nothing. show the default screen.
        } else if (selectedUrl.equals(ScreenNames.CATALOG_URL)) {
            nextScreen = ScreenNames.CATALOG_SCREEN;
        ...
        } else if (selectedUrl.equals(
                    ScreenNames.VALIDATE_BILLING_INFORMATION_URL)) {
            if (req.getSession()
                .getAttribute("shippingAddressRequired") != null) {
                boolean addrReqd = req.getSession()
                    .getAttribute("addrReqd").equals("true");
                if (addrReqd)
                    nextScreen = ScreenNames.ENTER_SHIPPING_INFO;
                else
                    nextScreen = ScreenNames.CONFIRM_SHIPPING_INFO;
            }
        }
        return nextScreen;
    }
}
```

Code Example 10.18 ScreenFlowManager

10.6.8 Model-View Synchronization

Following the MVC architecture, views implemented by JSP pages and JavaBeans components present data owned by their associated models implemented as enterprise beans. In the sample application, each Web-tier JavaBeans component serves as the view, with corresponding EJB-tier classes representing the model. Whenever

a model changes, it notifies interested views so that the views can update its presentation of the model.

In the sample application, the notification process is managed by `ModelUpdateManager` and `ModelManager`. `ModelUpdateManager` is responsible for converting a business event, such as `AccountEvent`, to a list of names of models that have changed due to this event. `ModelManager` uses this list to notify all views that have registered interest in the changed models to fetch the models' data.

The functions of `ModelManager` and `ModelUpdateManager` and their interactions with controller objects are described in the following sections.

10.6.8.1 Model Manager

`ModelManager` extends `ModelUpdateNotifier`, which provides methods for adding listeners of model change events and causing listeners (that is, views) to perform an update when a change event is received. `ModelManager` adds methods that create and return instances of view classes.

Code Example 10.19 presents excerpts from `ModelManager`. Note that `ModelManager` maintains references to both a `ServletContext` and an `HttpSession`. These objects in turn contain references to view objects (highlighted in the example). View objects specific to a client (for example, `AccountModel`) are maintained by an HTTP session object. View objects that can be shared by all clients (for example, `CatalogModel`) are maintained as an attribute of the servlet generated from `Main.jsp`.

```
public class ModelManager extends ModelUpdateNotifier {
    private ServletContext context;
    private HttpSession session;
    private ShoppingClientController sccEjb = null;
    private ShoppingCart cartEjb = null;
    private Account acctEjb = null;

    public void init(ServletContext context,
         HttpSession session) {
        this.session = session;
        this.context = context;
        getAccountModel();
    }

    public CatalogModel getCatalogModel() {
```

```
            CatalogModel catalog = (CatalogModel)
                context.getAttribute(WebKeys.CatalogModelKey);
            if (catalog == null) {
                catalog = new CatalogWebImpl();
                context.setAttribute(WebKeys.CatalogModelKey, catalog);
            }
            return catalog;
        }
        public AccountModel getAccountModel() {
            AccountModel acct = (AccountModel)
                session.getAttribute(WebKeys.AccountModelKey);
            if (acct == null) {
                acct = new AccountWebImpl(this);
                session.setAttribute(WebKeys.AccountModelKey, acct);
            }
            return acct;
        }
        ...
    }
```

Code Example 10.19 ModelManager

10.6.8.2 ModelUpdateManager

`ModelUpdateManager` is responsible for converting an `EStore` event to a list of names of models that have changed due to this event. Code Example 10.20 presents excerpts from `ModelUpdateManager`.

```
public class ModelUpdateManager {
    ...
    public Collection getUpdatedModels(EStoreEvent ese)
        throws RemoteException {
        ArrayList modelList = new ArrayList();

        if (ese instanceof CartEvent) {
            modelList.add(JNDINames.CART_EJBHOME);
        } else if (ese instanceof AccountEvent) {
            modelList.add(JNDINames.ACCOUNT_EJBHOME);
        } else if (ese instanceof OrderEvent) {
```

```
            modelList.add(JNDINames.ORDER_EJBHOME);
            modelList.add(JNDINames.INVENTORY_EJBHOME);
            modelList.add(JNDINames.CART_EJBHOME);
        } else if (ese instanceof LoginEvent) {
            modelList.add(JNDINames.ACCOUNT_EJBHOME);
        }
        return modelList;
    }
```

Code Example 10.20 ModelUpdateManager

10.7 MVC Summary

Figure 10.12 summarizes the references between the view, model, and controller classes.

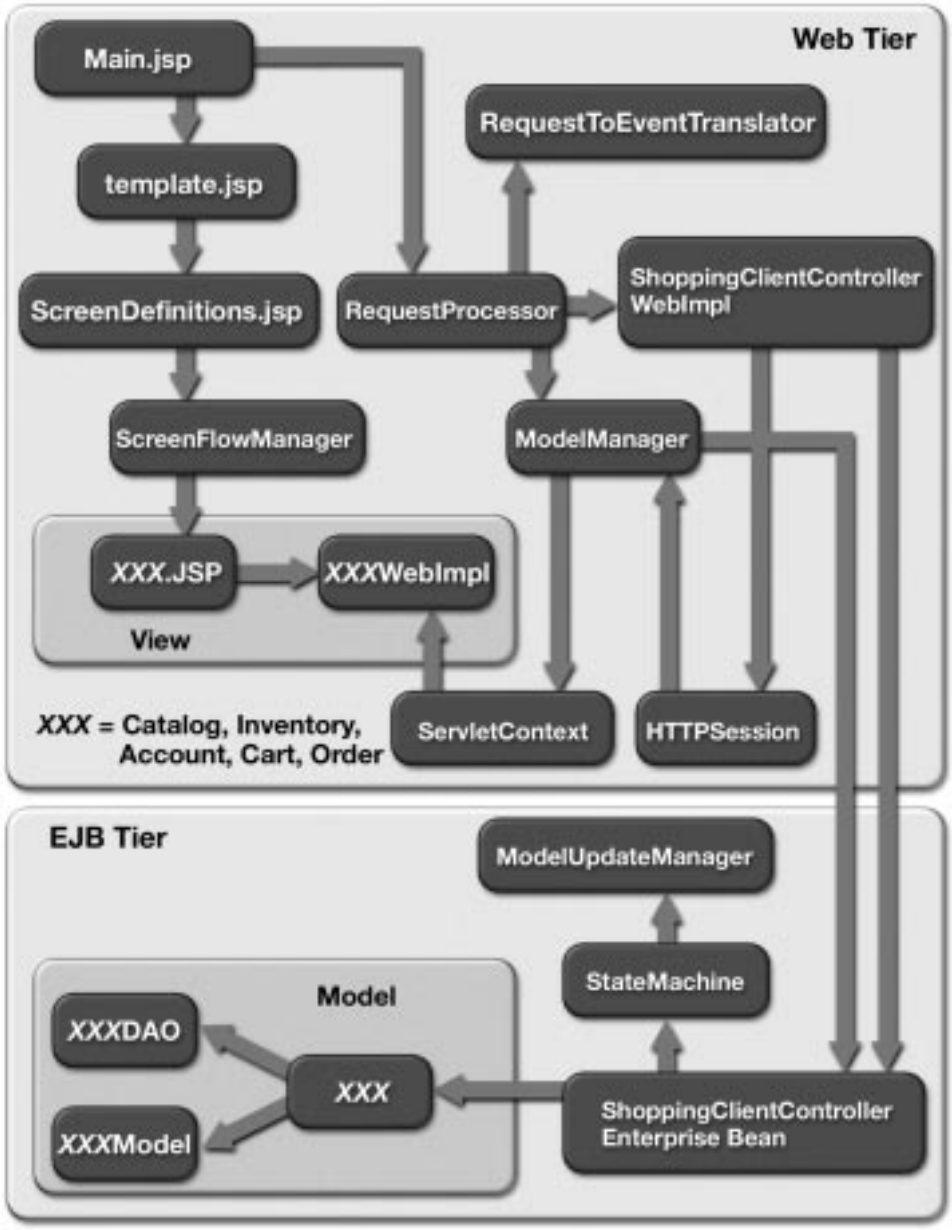

Figure 10.12 Object Reference Diagram

10.8 Stateless Services

The sample application uses stateless session beans for shared service objects. For example, in an e-commerce application, you might want to send order confirmation mail to customers on successful completion of an order. Such a service can be shared by all clients of the application. The sample application `Mailer` service objects are stateless session beans.

10.8.1 Example: A Mailer Bean

When a client places an order, an order event is passed to `ShoppingClientController`. Although the `handleEvent` method is defined by `ShoppingClientController`, `ShoppingClientController` delegates its implementation to a helper class named `StateMachine`. `StateMachine` interacts with `Inventory`, `Order`, and `Mailer` enterprise beans to debit the quantity of the purchased item, insert the order details, and finally send the confirmation email to the client.

The last thing that `StateMachine` does in the `createOrder` method is to send an order confirmation message. It does this by first creating `Mailer` stateless session bean and then invoking the `Mailer.sendOrderConfirmationMail` method (shown in Code Example 10.21). This `method` uses the order ID to obtain the information needed for the confirmation message from the `Order` and `Account` entity beans. The `Mailer` then invokes the `createAndSendMail` method of its helper class `MailHelper`.

```
public void sendOrderConfirmationMail(int orderId) throws RemoteEx-
ception {
    OrderDetails orderDetails = null;
    try {
    OrderHome orderHome = EJBUtil.getOrderHome();
    Order order = orderHome.findByPrimaryKey(orderId);
    orderDetails = order.getOrderDetails();
    } catch (FinderException fe) {
        ...
        return;
    }
    String userId = orderDetails.getUserId();
    AccountDetails acctDetails = null;
    try {
        AccountHome acctHome = EJBUtil.getAccountHome();
```

```
            Account acct = acctHome.findByPrimaryKey(userId);
            acctDetails = acct.getAccountDetails();
        } catch (FinderException fe) {
            ...
        }
        String subject = "Your order# "+orderId;
        String HTML Contents =
            "This message is a confirmation of your order# "
            + orderId + ". Please save it for your records.";
        getMailHelper().createAndSendMail(acctDetails.
            getEmail(), subject, HTML Contents);
    }
```

Code Example 10.21 `Mailer.sendOrderConfirmationMail`

Code Example 10.22 illustrates the `createAndSendMail` method of `MailHelper`. This method looks up a mail session in the JNDI namespace, creates a MIME message, sets the mail headers, collects the contents of the message into a string, and then sends the message.

```
public void createAndSendMail(String to,
    String subject, String HTML Contents) {
    try {
        InitialContext ic = new InitialContext();
        Session session = (Session) ic.
            lookup(JNDI Names.MAIL_SESSION);

        // construct the message
        Message msg = new MimeMessage(session);
        msg.setFrom();
        msg.setRecipients(Message.RecipientType.TO,
                              InternetAddress.parse(to, false));
        msg.setSubject(subject);
        collect(subject, htmlContents, msg);
        msg.setHeader("X-Mailer", "JavaMailer");
        msg.setSentDate(new Date());
        // send the message
        Transport.send(msg);
    } catch (Exception e) {
```

```
            ...
        }
    }
```

Code Example 10.22 MailHelper.createAndSendMail

10.9 Deployment

Much of the behavior of the sample application is determined by information specified in its deployment descriptors: `estore_ejb.xml`, `estore_ejbruntime.xml`, `estore_war.xml`, and `estore_warruntime.xml`. Elements specified in these deployment descriptors are discussed in detail in section 7.1 on page 173 and section 7.3 on page 183.

10.10 Transactions

The sample application's persistent data is stored in two databases: `eStoreDataSource` and `InventoryDataSource`. The `eStoreDataSource` database holds information about accounts and orders. The `InventoryDataSource` database holds information about products, product categories, and the inventory of each product. When an order is placed, `ShoppingClientController` must access both databases. The sample application uses J2EE SDK support for distributed transactions to reduce the inventory of ordered products and add a new entry to the order table in an atomic operation. Note that a J2EE product is not required to support access to multiple JDBC databases within a single transaction. However, some J2EE products might choose to provide these extra transactional capabilities.

Recall that `ShoppingClientController` delegates the implementation of order processing to a helper class named `StateMachine`. `StateMachine` is responsible for maintaining consistency among the database tables represented by the enterprise beans that it calls. When `StateMachine` handles an order event it invokes the method `createOrder` illustrated in Code Example 10.17. For each bean, `StateMachine` gets a reference to the home interface and then creates an instance of the bean. `StateMachine` loops through the list of items in the order, finds the appropriate inventory item, and updates the `inventory` table for that item. Simply creating an instance of the `Order` bean causes an entry to be added to the `order`, `lineitem`, and `orderstatus` tables.

Note that neither the `StateMachine.createOrder` nor individual bean operations explicitly invoke transactions, because `ShoppingClientController` uses container-managed transactions. As a result, all database operations invoked by `ShoppingClientController` are automatically wrapped in a transaction by the container. The transaction context is automatically propagated to any enterprise beans that `ShoppingClientController` invokes (in this case `Inventory` and `Order`).

10.11 Security

This section describes the sample application's security requirements and discusses the ways these requirements are addressed using the J2EE security framework.

10.11.1 Requirements

The pet store application is designed to be deployed on the Internet. Like many Web-based e-commerce applications, it allows anyone to interact with and use the application. Any user, regardless of whether they're a registered customer, can point a browser at the start URL of the application and browse through the catalog, viewing items, prices, inventory status, and so on. We call this *tire kicking*, and this class of users *tirekickers*.

A new customer can *sign up* using a form presented by the application. Once a customer has signed up, the customer can *sign in*, by providing a user name and password to the application. Only customers who have signed in are allowed to place orders and view order status. When an order is placed, the payment details, including the credit card number, must be transmitted in a secure manner.

Some users of the pet store application may receive special treatment. For example, a frequent shopper may a *preferred customer*, able to receive discounts or awards not available to normal customer. Another class of special user might be system administrators, with unlimited access to information on the site. For example, they might be able to fetch a list of all orders placed after a certain date.

These high-level functional requirements translate into the following security requirements:

- **User Authentication**

  Users of the pet store application can be either authenticated or unauthenticated. The user must be authenticated to access a protected resource. The

application should be able to identify, differentiate, and be able to make access control decisions based on this distinction.

There should be a way to associate each authenticated user with one or more categories. For example, user `A` could be recognized as a customer, while user `B` could be recognized as a preferred customer.

- **Authorization**

  The application associates permissions with resources such as Web pages or enterprise bean methods. Examples of the kind of authorization constraints the application should allow:

  - Anyone (authenticated or not), to see the URL `/control/product`, or invoke the `getProducts` method of `Catalog`
  - All authenticated users to see the URL `/control/placeorder`
  - Only preferred users to see the URL `/control/discounts`

- **Confidentiality**

  Some user information, such as a credit card number, must be transmitted confidentially to the application.

- **User Administration**

  The sample application has its own set of users. This set of users grows when new users add themselves using a Web-based interface. Note that other applications, such as those developed for in-house use within an enterprise assume and use the set of users defined in the operational environment. The sample application does not depend on the operational environment to get its set of users.

10.11.2 Implementation

The pet store application uses many features of the J2EE platform to address its security requirements in a simple and transparent manner. By design, security in the J2EE platform is mostly declarative. In some places however, we make security decisions in our components programmatically, because we needed to make authorization decisions based on the content or state of the object.

10.11.2.1 User Authentication

A J2EE application must be capable of authenticating users that access the application from a variety of clients. This section describes how the pet store application authenticates users of the shopping interaction Web client, how it could authenticate users of an administration application client, and how it handles unauthenticated users.

10.11.2.1.1 Web Client Authentication

Most of the interactions with the sample application occur through the Web-based interface. Form-based authentication, one of the standard authentication mechanisms in the J2EE architecture, is used to authenticate these interactions.

In form-based authentication, a Web container designates an application-specific page containing an HTML form for logging in. The sample application uses the page `login.jsp` as this page. This page contains an HTML form that prompts for a user name and password and is displayed when the user tries to access a resource that has been designated as being protected. The sample application also uses form-based authentication to enable:

- Explicit signin

  The sample application allows users to explicitly sign in by clicking the Sign-in link in the user interface. The Sign-in link points to `/signin` which is a dummy URL that is inaccessible to unauthenticated users.

  When the user clicks Sign-in, the application attempts to take them to the `signin.jsp` page, which is denied since the page is protected. As a result, the `login.jsp` form is shown instead.

  Note that we cannot simply make the Sign-in link point to `login.jsp` because an authorization failure must occur for the form-based authentication mechanism to be activated.

- Informing the user about failed authentication

  The sample application retries the protected resource after authentication through form-based authentication irrespective of the outcome of authentication. If authentication failed, the `login.jsp` form will be shown to the user again. At this point, it is desirable to do two things: first, make sure that the form comes back already filled with the values that were posted in the last try, and second, inform the user somewhere in the form that authentication failed the first time.

Form-based authentication does not provide a portable way to return the form to the user with the values posted in the failed try. The `POST` to the form is handled by the Web container, and never returned to the form.

Informing the user that sign on failed is easier to accomplish. To do so, the `login.jsp` page uses a session-scoped bean (see Code Example 10.23) stored each time the form is accessed. If this time is *close* to the current time, and the current request is unauthenticated, then the sample application prints a message indicating that the login failed earlier. A request to `login.jsp` would always be unauthenticated, unless there is an application programming error. The only situation where `login.jsp` is shown should be when the request is unauthenticated. Using a similar mechanism, it is also possible to go to an error page after a fixed number of retries.

```
<jsp:useBean id="last_login" class="...">
<% if (last_login.getTime() - currentTime < ... {
   %><font color="red">Login failed, try again<p></font><%
} %>
```

Code Example 10.23 login.jsp

- Abandoning signin

  Sometimes the `login.jsp` page comes up because the user tried to access a protected resource. If the user does not have an account and needs to create one, the application should abandon the login process and start user signup instead.

  The pet store application's `login.jsp` page has an additional button called `New User`, to let new users sign up before they attempt to sign in.

  The J2EE platform maintains information about the state of the signin in the `HttpSession` and times it out when the login attempt is abandoned.

- Treating newly created users as signed in

  When a new user *signs up* they should be treated as *signed in* for the duration of that session. This is the case for the sample application, which manages its own set of users. However, in the J2EE architecture, this is not the case. The only way users can sign in is through form-based authentication. Since form-based authentication is not invoked when a user signs up, they still need to explicitly sign in order to treat be treated as an authenticated user.

  The sample application uses a non-portable, private API provided by the

J2EE SDK to achieve the desired results.

- Detecting user login

  The form-based authentication mechanism is designed to be transparent. However there are cases where we want to be aware of the *first* request that the user makes after they have signed in. For example, in the sample application, the fetching and caching of user profile information is triggered when the user logs in.

  Since the `POST` to the login form is processed by the platform, there is no direct way of doing this. We use the code shown in Code Example 10.24. `RequestProcessor` detects when the user logs in and fires a `LoginEvent` which can then be handled to get the desired effect.

```
private void checkForWebServerLogin(HttpServletRequest req){
    if ((req.getUserPrincipal() != null) &&
        !req.getUserPrincipal().getName().equals("guest") &&
        !mm.getAccountModel().isLoggedIn()) {
            EStoreEvent loginEvent = null;
            loginEvent = eventTranslator.createLoginEvent(req);
    ...
```

Code Example 10.24 Triggering the LoginEvent

Let's look at the condition being tested. We first check if the principal is set on the current call. If it is, it means that some user is currently logged in. Next we check if our account bean knows about it. If `accountBean.isLoggedIn` returns false, it means that the account bean is not aware of the login yet. This is exactly the condition when we want to trigger the login event. Once the login event is processed, `account.isLoggedIn` would return true.

10.11.2.1.2 Application Client Authentication

In the J2EE platform, stand-alone clients are authenticated by an application client container. The application client may authenticate its user in a number of ways. The techniques used are platform-dependent and not under control of the application client. The application client container may integrate with the platform's authentication system, providing a single signon capability. The application client container may authenticate the user when the application is started. The application client container may use lazy authentication, only authenticating the user when it needs to

access a protected resource. The J2EE specification does not describe the technique used to authenticate the user.

The J2EE SDK generates a client JAR file[1] when enterprise beans are deployed. This library contains stub classes for accessing enterprise beans as well as a mechanism provided by the J2EE SDK for handling authentication to an EJB server.

10.11.2.1.3 Handling Unauthenticated Users

The sample application allows for anonymous, unauthenticated users to access the application and browse selected features of the pet store. Even in such cases, calls to the EJB tier must specify a valid principal; the EJB container rejects all calls without a security principal. That is, if the user invokes a feature that tries to call the EJB tier without authentication, the EJB container will not let the call go through.

However, since the Web interface needs to support anonymous, unauthenticated users, the J2EE platform defines a mechanism to do so. The responsibility for ensuring that unauthenticated calls are made using some principal is delegated to the EJB *client* container. In the sample application, the Web container performs this role by:

- Associating the credentials of a *special* user called `guest`[2] to an unauthenticated user when an EJB method is invoked.
- Treating unauthenticated users as follows:
  - The `getUserPrincipal` method of the servlet API returns `null` for such users.
  - The form-based or other authentication mechanism will be activated when a protected Web resource is accessed.

1. This is specific to the J2EE SDK. Other J2EE products may provide other mechanisms to create client JAR files.

2. We call it `guest` here. It may be called different things in different implementations. The important thing to note however is that it is different from any valid user of the system and is treated specially at deployment.

10.11.2.2 Authorization

Sample application security is specified in terms of the security roles `customer` and `gold_customer`.

- A `customer` is a registered user of the application. Users in the `customer` role can place orders and complete purchases. In the current release of the sample application, the default user `j2ee` is in the `customer` role.

- A `gold_customer` is a customer with special privileges. Additional awards are available to them. In the current release of the sample application, all users that sign up are assigned to the `gold_customer` role.

By default, the J2EE SDK assigns the `ANYONE` role to an enterprise bean method. The `guest` user, which is anonymous and unauthenticated, is assigned to the `ANYONE` role.

In the sample application, access to the URLs `/control/signin` (described in section 10.11.2.1 on page 311) and `/control/placeorder` is restricted to the roles `customer` and `gold_customer`. The `security-constraint` declaration for `/control/signin` is shown in Code Example 10.25.

```
<security-constraint>
    <web-resource-collection>
        ...
        <url-pattern>/control/signin</url-pattern>
        <http-method>POST</http-method>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>no description</description>
        <role-name>gold_customer</role-name>
        <role-name>customer</role-name>
    </auth-constraint>
    ...
</security-constraint>
```

Code Example 10.25 Security Constraint Declaration

In the current release, the sample application does not limit enterprise bean method invocation to specific security roles.

10.11.2.3 Confidentiality

Confidentiality constraints are specified at deployment time by setting the `transport-guarantee` element in the Web component's deployment descriptor to `CONFIDENTIAL`. In the current release, the sample application doesn't demonstrate confidentiality mechanisms.

10.11.2.4 User Administration

Many applications will need to perform two tasks that aren't handled by the J2EE platform: managing user profile information (other than security credentials and attributes) and adding new users to the system dynamically.

10.11.2.4.1 Maintaining User Profiles

In addition to keeping security credentials, the sample application needs other information about the user's preferences and personalization. The J2EE security framework will keep the security credentials, such as the user name and password, as well as attributes such as the set of roles that the user belongs to. The sample application needs another mechanism to maintain additional information for a user.

To do so, it maintains a separate relational table for user profile information. This table is called the `accounts` table, and is accessed through the `Account` enterprise bean. The user name is unique for each sample customer, and we use it as a key to the accounts database. Code Example 10.26 shows how the `getCallerPrincipal` method is used to retrieve the `userId` of the user making the current enterprise bean method call. The value returned from this method is used as a key to retrieve profile information for the user.

```
public Account getAccount() {
    if (acct == null) {
        try {
            String userId = sc.getCallerPrincipal().getName();
            AccountHome home = EJBUtil.getAccountHome();
            acct = home.findByPrimaryKey(userId);
        } catch (FinderException fe) {
            ...
        }
```

```
            catch (RemoteException re) {
                throw new EJBException (re);
            }
        }
        return acct;
    }
```

Code Example 10.26 ShoppingClientControllerEJB.getAccount

10.11.2.4.2 Adding New Users

The J2EE platform does not standardize a mechanism to add users dynamically to applications. Any application that requires this feature needs to do so in a non-portable, container-specific manner.

In such a situation, it makes sense to isolate all the non-portable code in one place. This small piece of platform-specific code can later be replaced if the application needs to be ported to a different container implementation.

The J2EE SDK provides a container-specific API for managing users based on the concept of realms. A realm is a collection of users under the same authentication policy. An application can provide its own realm and plug it into the J2EE SDK for the container to use for authentication, or it can use realm API methods such `addUser`, on the existing default J2EE realm.

The sample application uses the default J2EE realm. It uses the `addUser` method of the realm to add new users while processing the `signup.jsp` form.

In addition to specifying the user name and password of the user being added, we also need to specify the roles that this new user can assume. This is achieved through the `addUser` method, which takes an array of roles as an argument.

10.11.2.5 Programmatic Security

The J2EE platform encourages the use of declarative security. However there are places where one needs to make access control decisions based on the current state of the system. Such decisions must be made by programmatically encoding their rules in the application.

The J2EE platform allows the application to identify the principal making the call as well as the role that the caller is in, in both the Web and EJB tiers. The sample application uses these facilities as follows.

10.11.2.5.1 Web Tier

In the Web tier, the sample application uses the `getUserPrincipal` and the `isUserInRole` methods as follows:

- `getUserPrincipal`: This method is used to get the ID of the user that connects to the application. This user ID could be used in the template to print the user ID in the banner as part of a welcome message. Another way that the sample application uses this information is to determine if a user is logged in. Code Example 10.24 illustrates this use.
- `isUserInRole`: In the current release, the sample application doesn't use this method. This method could be used in the template in order to show a different icon based on whether the user is a preferred or a regular customer. Additionally, there might be special items that we only show to preferred customers. Thus the catalog can be filtered based on the result returned from calling this method with the `gold_customer` role.

10.11.2.5.2 EJB Tier

In the EJB tier, the sample application uses `getCallerPrincipal` and `isCallerInRole` methods as follows:

- `getCallerPrincipal`: This method is used to get the caller key to be able to access profile information associated with the principal associated with the call.
- `isCallerInRole`: This method is used by the order processing module to enforce award rules based on whether the customer is a preferred customer. Code Example 10.27 illustrates the use of `isCallerInRole`.

```
private int getBonusMiles() {
    int miles = (totalPrice >= 100) ? 1000 : 500;
    if (context.isCallerInRole("GOLD_CUSTOMER"))
        miles += 1000;
    return miles;
}
```

Code Example 10.27 OrderEJB.getBonusMiles

Notice the use of the embedded role name GOLD_CUSTOMER. When role names are embedded in the code, the Application Component Provider needs to identify these roles in a deployment descriptor so that the Deployer can ensure that they are mapped correctly when the application is deployed. Code Example 10.28 shows the portions of the sample application deployment descriptor where this happens.

```
<security-role-ref>
    <role-name>GOLD_CUSTOMER</role-name>
    <role-link>gold_customer</role-link>
</security-role-ref>
...
<assembly-descriptor>
    <security-role>
        <role-name>gold_customer</role-name>
    </security-role>
    ...
</assembly-descriptor>
```

Code Example 10.28 Deployment Descriptor Element for Embedded Roles

In this excerpt from the deployment descriptor, the Application Component Provider declares the use of GOLD_CUSTOMER in the application using the security-role-ref element. The Deployer must ensure that this role is linked to the gold_customer security role.

10.12 Summary

This chapter illustrates the J2EE programming model in the context of an in-depth description of a multitier Web application: the pet store e-commerce application.

The functionality of the sample application was determined using a scenario-driven approach. Walks through scenarios illustrated the requirements for the user interaction as well as the interactions that happen *within* the system. Analysis of the sample application identified three very different kind of interactions: a shopping interface that allows shoppers to buy items online, an administration interface for carrying out store administration activities, and a business-to-business

interface through which the store can interact with suppliers. The discussions in this chapter focused mainly on the shopping interactions.

The architecture of the sample application partitions its functionality into modules, assigns functionality to tiers, and decomposes the modules into specific objects to represent the behavior and data of the application. The principles guiding the architecture include reuse of software designs and code, separation of stable from volatile code, object decomposition along skill lines, and ease of migration from a Web-centric to EJB-centric model.

The architecture of the sample application adapts the Model-View-Controller design pattern to the domain of enterprise applications. The model represents the application data and the business rules that govern access and modification of this data. The view renders the contents of a model. It accesses data from the model and specifies how that data should be presented. The controller defines application behavior; it interprets user gestures and maps them into actions to be performed by the model. In a stand-alone GUI client, these user gestures could be button clicks or menu selections. In a Web application, they appear as `GET` and `POST` HTTP requests to the Web tier. Based on the user gesture and the outcome of the model commands, the controller selects a view to be rendered as part of the response to this user request.

The J2EE platform provides system services that simplify the work that application objects need to perform. The sample application uses the Java 2 SDK, Enterprise Edition support for distributed transactions across multiple JDBC databases. In addition, it uses deployment and security capabilities of the J2EE platform to support customers with different profiles.