Programmer to Programmer ™

J2EE Design Patterns Applied

Real World Development with Pattern Frameworks

Craig A. Berry, John Carnell, Matjaz B. Juric, Meeraj Moidoo Kunnumpurath, Nadia Nashi, Sasha Romanosky

wrox
Programmer to Programmer

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com

What you need to use this book

To run the samples in this book, you will need to have the following:

- A J2EE 1.3 server implementation. All the code in this book was tested on the Sun J2EE 1.3 Reference Implementation.
- A relational database. All the code in this book was tested on MySQL.

The book assumes that you are familiar with the development and deployment of J2EE components such as servlets, JSP pages and EJBs.

Summary of Contents

Patterns Applied to Manage Security

In this chapter, we introduce security patterns and their benefits throughout the design of our case study, a J2EE Web Banking application. We will define the scope and requirements of this application, identify relevant security patterns, and apply them to the design of both the application and its operating environment. We will develop use cases, and finally present code for a few of the major Java classes.

What are Security Patterns?

Security patterns provide techniques for addressing known security issues, in the same manner as J2EE and other object-oriented patterns provide proven techniques for solving known programming problems.

Security patterns work together to form a collection of best practices, whose ultimate goal is to support an organization's security policy – a policy that addresses not just application security, but host and network security, as well. Thus they can (and ideally should) be applied to the design and development of applications, and to the configuration and management of the hosts, and the network within which these applications operate. Security patterns, however, do not define specific technologies, coding styles, or programming languages. They do not identify industry vendors, application version numbers, or patch levels.

Benefits of Using Security Patterns

Similarly to standard object-oriented patterns, security patterns provide the following benefits:

- They can be revisited and implemented at anytime to improve the design of a system
- Less experienced practitioners can benefit from the experience of more advanced practitioners
- They provide a common language of discussion, testing, and development
- They can easily be categorized, searched, and refactored
- They provide reusable, repeatable, and documented security practices

When to Use Security Patterns

Security patterns can provide guidance when dealing with the following issues:

- Whenever data is being sent to or received from an external system, application, or object:
  - Will it validate the information based on length, value, or type?
  - Is the communication channel secure? Does it need to be?
  - What is the origin of the data, is it a trusted or non-trusted source?
- Whenever an application is accessible by trusted or non-trusted users:
  - Who is trying to access the application?
  - Is their request legitimate? Should it care?
  - Does it know how to process their request? What should it do if it doesn't?
  - Does it know about every attempt to access the system? How can it be sure?
- Whenever data is considered confidential or sensitive:
  - How is the data being protected?
  - Are these means sufficient or unwarranted?
  - Is the data being stored or backed up elsewhere? Is this adequate?

Secure Programming

As mentioned previously, security patterns are essentially best practices and can assist in the design of secure applications. However, they are not a replacement for secure programming techniques. Following proper coding standards in all languages is essential for developing resilient software. The following are few examples of proper coding:

- Data validation
- Code, design reviews
- Scoping
- Synchronized operations
- Secure (dynamic) class loading
- Proper exception handling, error reporting, logging

The Web Banking Case Study

The goal of this case study is to apply security patterns to the design of a web banking application. The application will be a J2EE web-based one, which will act as a front end to an existing banking system. We will identify the features of the application, and then define the key business and technical requirements.

High-level Overview

The Wrox Bank is a national bank with branch offices and ATMs (Automated Teller Machines) located across the country, and operates on an existing computing (mainframe) infrastructure. The bank is being pressured by customers to provide banking services online. A recent survey revealed the following three services as most important to the customers:

- View account balance
- View account activity
- Transfer funds between accounts

Assumptions

The following assumptions can be made about this case study:

- The existing banking infrastructure consists of a trusted mainframe system, which will be capable of supporting all activity generated by this web-based application
- Connectivity to the back-end mainframe occurs over a dedicated, high-speed network
- The creation of web-based accounts (including usernames and passwords) is performed at branch locations and is outside the scope of this online application

Business Requirements

Business requirements define the features or services of an application.

The Wrox Web Banking application will be web-based and accessible over the Internet by standard web browsers (wireless devices will not be supported at this time). It will be a front end to the existing banking infrastructure, that is, it will not duplicate the core account information of the mainframe.

There will be three types of users of this application:

- Anonymous users are those who access only the public pages of the web site and cannot log in and thus cannot perform any banking activities.
- Regular customers are those who perform the following activities:
  - Log in to the application: after successful form-based authentication, the application will create a user session, allowing the customer to access other services
  - Log out of the application: this will terminate the user session
  - View account balance: immediately after login, the application will display a list of the customer's active accounts and their balances
- Preferred customers are those who perform all the activities of Regular customers in addition to the following:

- View account activity: beginning with the most recent transaction, this will display transactions as of the last statement period (30 days). At this time, the customer will not have the ability to view activity previous to this period. The following table specifies the fields that will be displayed for each transaction:

Field	Description
Date	Specifies the time and date the transaction was processed by the back-end system.
Description	Provides a description of the transaction.
Type	This field should contain one of the following values: ATM, Cheque, Web, Service Charge, or Other. ATM implies that the transaction was processed at an ATM. Cheque implies that a credit or debit was made using a cheque. Service Charge implies that the bank made the transaction. Other specifies all other types of transactions.
Reference Number	Specifies the cheque number if the transaction type was a Cheque, otherwise the reference number (as generated by the back-end system).
Amount	Specifies the amount of money that was withdrawn (shown as negative value) or deposited (shown as a positive value).

- Transfer funds between active accounts: This service will allow Preferred customers to transfer money from one active account to another. There will be no limit on the amount of money that can be transferred as long as the sending account has sufficient funds to support the transfer. Funds transfer can only occur between two active accounts.

By default, only the primary chequing and/or saving account will be active. Other accounts, such as credit card or line of credit accounts will be made available (active) only at the request of the customer at a branch location. Customers will only be allowed to view account information for which they are authorized and will not have access to other customer's account information. Confidential information will not be sent unprotected over an untrusted network. User accounts will be locked until the next business day if an incorrect password has been entered three consecutive times within 24 hours. User sessions will expire after 10 minutes of inactivity.

Technical Requirements

Technical requirements, also known as non-functional requirements, define issues such as performance, scalability, and availability. The application consists of the following tiers, each of which will exist on a separate network segment protected with a firewall restricting all but essential network activity:

- Presentation (web): Consists of a load balancer and web servers
- Application: Supports a cluster of application servers
- Data tier: The user (data) store will contain at least the following information:
  - Customer ID: a unique customer identification number as assigned by the back-end mainframe system
  - Username: as provided by the application to the customer
  - Full name: first and last name as listed in the mainframe system

- Password: as provided to the customer by the application
- Role assignment: as assigned by the back-end system
- The list of active accounts: a list of those accounts accessible by the customer of the web banking application

The back-end banking system will be the authoritative source of data for all remaining customer information including (but not limited to) the following:

- Personal customer information (full name, address, and so on.)
- Customer account numbers
- Customer account balances and history

All confidential information sent to the customer over HTTP will be transmitted in a secure fashion using 128 bit SSL. The web-based infrastructure will support high availability (HA) via load-balancing, clustered web, application, and database servers. Each network device and application will be hardened and configured with the latest patches. A testing and staging environment will be configured to match the production environment as closely as possible. Only authorized personnel will have access to production, and staging servers.

Security Patterns

Let us now identify and discuss security patterns as they apply to this case study. Given the scope, business, and technical requirements of this application, we will identify relevant security patterns that will help us develop a more secure application.

The Single Access Point Pattern

The Single Access Point pattern describes a system with a single point of entry; an entry that is common to all incoming requests, and is the only way to gain an entry into a system. All users (or other applications) requesting access must first pass through this entryway. By employing the Single Access Point pattern, the application is assured (as best it can) that any given user has achieved basic authentication and not bypassed any identification checkpoints. For example, unauthorized requests for protected resources can be automatically redirected to this access point for proper validation.

This pattern is typically represented by a single login prompt to an organization's network or individual server. Another example is an application that provides a single login page versus separate login prompts for each service.

Larger, more complex applications can benefit more from this. J2EE applications, for example, by virtue of their extensibility, tend to incorporate many disparate applications such as messaging, rules engines, and reporting applications, each of which may require some form of user authentication. Rather than developing login pages for each ancillary application, the Single Access Point is used to gather user information once, and forward it along.

The J2EE security model provides a simple mechanism to implement the Single Access Point pattern. By employing declarative security in the J2EE web tier, the designer is able to specify the web resources (URLs, URL patterns, and HTTP methods) that are to be protected. When an anonymous user requests a protected resource, the application (web container) will attempt to authenticate the user via a number of mechanisms. The J2EE platform natively supports basic authentication, forms-based authentication, and client-side certificates. Our Wrox Web Banking application employs forms-based authentication.

The Check Point Pattern

The Check Point is a pattern that centralizes and enforces authentication and authorization. It is the responsibility of this mechanism to determine if a user has sufficient privileges to grant access to a requested resource. By centralizing this logic, the Check Point pattern also affords easier management of application policies and business rules. Designers are able to modify and extend these rules without altering the remaining application. The Check Point pattern can apply to any of the following situations:

- Consider a system with multiple security levels. To initially gain access, a user enters a basic username and password. The Check Point grants them access to all resources matching this level of security. To achieve higher security clearance and access more sensitive information, the user would be required to provide stronger credentials such as a digital certificate. Another stronger authentication mechanism could be biometric identification. The Check Point pattern manages the security requirements of the resources, determines their security level, and provides the users access according to predefined security policies.
- Consider a system with multiple user stores, where the authentication credentials of some users are stored in an LDAP directory, some in a RDBMS, and others still in a mainframe system. It is the function of the Check Point to communicate with, and validate users against, any and all of those user stores.
- Consider an application that is currently only accessible via HTTP browsers but must extend services to end-user devices like IVR, cellular phones, or PDAs. A gateway can be created to forward requests from these devices to the application but the security will be assured if the requests from the devices are cleared through the Check Point.

Note that a simple application may only employ one or two of these scenarios in a single Check Point whereas a much more complex system may have multiple Check Points and may use each of these scenarios multiple times. For example, a multi-national organization may employ a Check Point at each national border to enforce the security policy of that country. Even then, it may incorporate regional Check Points to support local end-user devices.

Much of the functionality of the Check Point pattern can be supported natively in the J2EE security architecture with the Java Authentication and Authorization Service (JAAS). JAAS provides both pluggable and stackable authentication. The pluggable authentication module (PAM) framework abstracts the application from the underlying authentication code, thereby removing dependencies and allowing for greater flexibility and selection of authentication mechanisms.

PAM also provides stackable authentication that defines where and when any particular mechanism is 'optional', 'required', or 'sufficient'. In the above discussion of a system with multiple security levels, a user's successful authentication would be 'required' to achieve higher security clearance. To access resources of lower security levels, no further authentication would be required because they already possess 'sufficient' privileges.

The Role Pattern

The Role pattern describes the disassociation of a user from their privileges. A user is assigned one or more roles and each role is granted one or more privileges. A user can be assigned to, or removed from, a role whenever their responsibilities change and roles can be granted new privileges as new resources or objects become available. This approach provides two very important benefits:

- No modification to the underlying access control objects is necessary when updating role definitions or user privilege assignment
- Permissions are more intuitive to manage since roles closely represent a company's organizational structure or the types of users of an application like developer, administrator, manager, Silver, Gold, or Platinum Customer, and so on

More sophisticated implementations of the Role pattern, commonly known as Role-Based Access Control (RBAC) extend the basic Role pattern by applying the following concepts:

- Inheritance (or Role Hierarchy): when the sum total of privileges granted to a role, equals the privileges for that role as well as privileges of lesser, associated roles.
- Separation of duties: Static, Dynamic, and Organizational separation of duties represent real-world conditions where individuals cannot belong to two roles at the same time. Additionally, they cannot belong to a given set of roles. For example, a bank manager cannot also be an auditor or a bank customer cannot also be a teller.

The business requirement for the Wrox Web Banking application has identified three roles; Anonymous User, Regular Customer, and Preferred Customer, each with their associated privileges. Specifically, an Anonymous user can view all public pages. A Regular Customer has access to all public pages and can view their account balances. A Preferred Customer has all the privileges of the Regular Customer plus the ability to transfer funds between their accounts and view their account history.

The J2EE platform incorporates roles into its architecture by way of a declarative security model. Simple role-based access control can be achieved by defining roles (in declarative syntax) within the `ejb-jar.xml` and `web.xml` deployment descriptors. When a protected resource is requested, and the user is authenticated, the web container or enterprise bean references the role declarations and either permits or denies access to the web resource.

Risk Assessment and Management

Security equals risk management. Risk assessment and management speak of the "reasonable" and "appropriate" effort required to protect an application and its resources and are the first step in a security analysis. The goal is to perform a task in a secure manner, not to consume resources, over-engineer code, or unnecessarily encrypt publicly available information. It can be said that risk is proportional to the following thee factors:

- Threat: the frequency of attempts or successes
- Vulnerability: the likelihood of success
- Cost: cost of successful breach, or value of the resource

The greater any of these factors, the greater the overall risk will be. A proper risk assessment ensures that not only the application is being properly protected, but also each system with which it has direct or ultimately indirect contact. For example, the web servers of the Wrox Web Banking application are a target for attack, not because they necessarily contain sensitive information themselves, but because they can be used to launch a more rewarding or malicious attack against targets like the back-end banking system. (Of course, information could certainly be harvested from the web servers and used in other attacks.)

Clearly, the threat for these front-line applications is high. Their vulnerability depends on the extent to which the application has been securely written – it could be quite low. The cost of a breach in the application can include everything from the value placed on stolen code and customer data, to time lost detecting and repairing damage.

Authoritative Source of Data

When an application blindly accepts data from any given source then it is at risk of processing potentially outdated or fraudulent data. Therefore, an application needs to recognize which, of many possible sources, is the single authority for data. Understanding the authoritative source of data means recognizing where your data is coming from and knowing to what extent you can trust the validity of such information. In short, never make assumptions about the validity of unverified data or its origin.

In most cases, determining the authoritative source of data will lie with the owner of the business process. The owner understands better than the application designer or developer the purpose of the information in a larger context. Information security groups and application designers, however, should still advise the business owner on the volatility and integrity of the data source(s) under consideration.

While the interface of the Wrox Web Banking application may differ (from an ATM or teller), the account information is retrieved from a single data source – the back-end mainframe system. Attempting to duplicate customers' personal and account information locally would present unnecessary technical and procedural issues at this stage.

The banking application will therefore be designed to access the mainframe system for all customer account information. Only the information that is relevant to the web banking system, for example, web banking usernames, passwords, and role assignments need be stored in a local data store. The authoritative source of data for customer account information is the back-end mainframe, whereas the authoritative source of data for web banking usernames and passwords is the local user store.

Authoritative source of data also embodies the premise of validating information received from a user or system. In our case study, user input is the form, which contains the username, password, monetary amount; each of these data fields must be validated at least for character type and length. At a minimum, this validation should be performed on the server side by the processing servlet. Additionally, client-side validation (within JavaScript, for example) will offer a first check of the information before it is submitted to the server.

Wrox Web Banking Use Cases

Functional and technical requirements have been identified and a number of security patterns have been discussed. These will aid us in properly defining the use cases for this application.

Actors

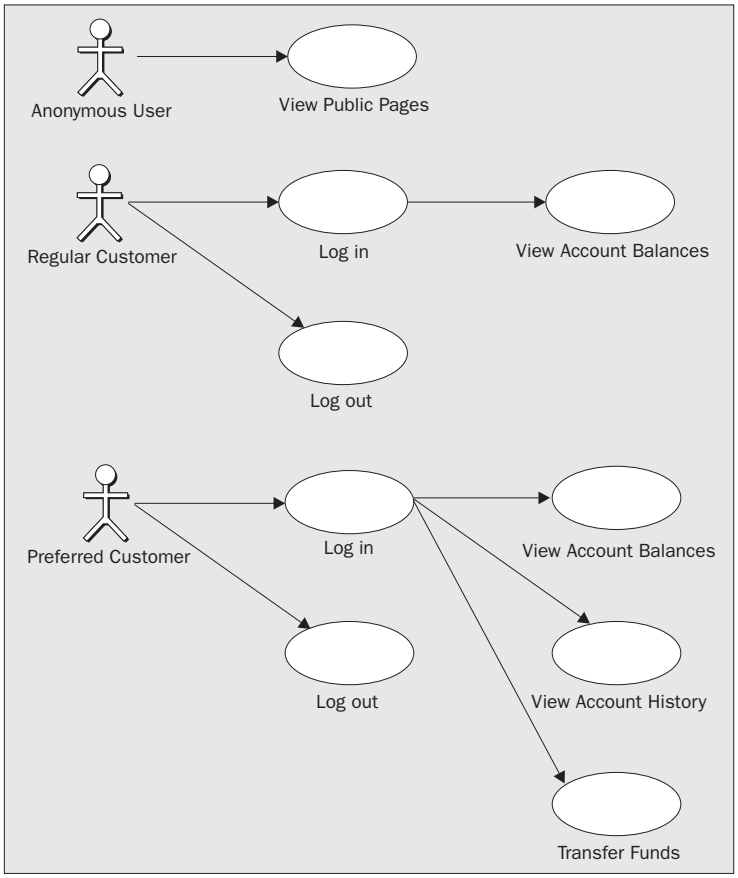
Based on the functional requirements, Anonymous users, Regular Customers, and Preferred Customers are the identified actors.

Use Cases

The following use cases have been identified:

- View public pages
- Log in to the application
- Log out of the application
- View account balances
- View account activity
- Transfer funds between Wrox accounts

Given the aforementioned use cases, consider the following use case diagram:



The diagram attempts to demonstrate how the Preferred Customer has access to all the services of the Regular Customer with the addition of View account activity and Transfer funds.

View Public Pages

This is the trivial use case, which describes any unauthenticated user browsing the public pages of the web site.

Main Flow of Events

- The public pages represent the sections of the web site where no authentication is required
- An anonymous user accesses the public web site

Login Use Case

The Regular or Preferred Customers utilize this use case when they wish to access any of the Wrox Web Banking services.

Main Flow of Events

The customer is prompted to enter a username and password. On successful validation of their credentials, a session is created:

- The customer is prompted for a username and password.
- The customer enters the username and password and submits the information.
- The application first verifies the username with the local user store.
- If the username is found, the application attempts to validate the password with the local user store. If the password is validated the application creates a user session.
- The application retrieves the following information from the local user store: username, first and last name, role designation, and list of active accounts, and stores this information in the customer's session object.

Alternative Flow of Events

- If the username is not found the following an error message is displayed: "Invalid username or password, please try again".
- If the password does not match, the following error message is displayed: "Invalid username or password, please try again". The system records this failed login attempt. If this was the third unsuccessful login attempt, the account is locked till next business day. (Note the same error message is displayed for both invalid username and passwords. This is done to prevent attackers from guessing usernames based on returning error messages, otherwise known as username harvesting.)
- If the application is unable to create a user session, the following message will be displayed, "We are unable to process your request". A more detailed error message is logged to the system log file for inspection.

Logout Use Case

A Regular or Preferred Customer uses this use case when they are done using the application.

Main Flow of Events

The use case starts when a customer has successfully authenticated to the application. The customer selects the logout button and exits the application:

- A customer selects the logout operation
- The application terminates the user session
- The customer is redirected to a page confirming that the logout was successful

Alternative Flow of Events

Ten minutes of inactivity lapses. At this time, the application terminates the user session.

View Account Balances Use Case

A Regular or Preferred Customer utilizes this use case to view the account balances of their active accounts.

Main Flow of Events

The use case starts when a customer has successfully authenticated to the application. The application retrieves and displays the account balances for all active accounts:

- The application verifies that the customer has a valid session. If not, the user is redirected to the login page.
- The application retrieves a list of the customer's active accounts from the user session object and verifies their role assignment.
- The application requests account balance information from the back-end system.
- If the user is a Preferred Customer the application will provide hyperlinks to view account history.
- The application displays the information to the customer.

Alternative Flow of Events

- If the customer does not have any active accounts the application will display the following message, "You do not currently have any active accounts".
- If the application is unable to retrieve a list of the customer's active accounts, the following error message will be displayed, "The application is currently unable to process your request, please try again shortly". A more detailed error message is logged to the system log file for inspection.

View Account Activity Use Case

The Preferred Customer utilizes this use case to view account activity.

Main Flow of Events

This use case starts when the Preferred Customer selects an account from the list of account balances. The application retrieves account activity and displays the information:

- The customer selects an account from the list of account balances.
- The application verifies that the customer has a valid session. If not, the customer is redirected to the login page.
- The application verifies that the customer is part of the Preferred role. If they are not, the following error is displayed, "You are currently not allowed to perform this function". A more detailed message is logged. The application requests the account activity from the back-end system. This information is displayed to the customer.

Alternative Flow of Events

- If the customer does not have any active accounts, the application will display the following message, "You do not currently have any active accounts".
- If the application is unable to retrieve the customer's account history, the following error message will be displayed, "The application is currently unable to process your request, please try again shortly". A more detailed error message is logged to the system log file for inspection.

Transfer Funds Use Case

The Preferred Customer utilizes this use case to transfer money between accounts.

Main Flow of Events

This use case begins when the Preferred Customer sees a list of account balances. The Preferred Customer selects to transfer funds between a sending and a receiving account. The application verifies that sufficient funds exist in the sending account. The amount is debited from the sending account and credited to the receiving account. The back-end system records the transaction and generates a transaction number:

- The customer requests to transfer funds between active accounts.
- The application verifies that the customer has a valid session. If not, the customer is redirected to the login page.
- The application verifies that the customer belongs to the Preferred role.
- The customer selects the sending and receiving account, and the amount of funds to transfer.
- The application verifies that sufficient funds exist in the sending account.
- The application sends a request to the back-end system to perform the transfer.
- The back-end system performs the transfer and returns a status code and optionally, a transaction record.
- The application displays a confirmation message and the transaction record to the customer.

Alternative Flow of Events

- If the user is not a Preferred Customer, the following error message is displayed, "You are currently not allowed to perform this function". A more detailed message is logged.
- If the Preferred Customer does not have sufficient funds in the sending account, the following message is displayed, "Sorry, you do not have sufficient funds to perform this transfer".
- If the back-end system encountered an error and returned an error message, that message will be displayed and a more detailed message is logged.

Implementing the Case Study

In the next few sections, we will have a look at implementing the case study, using standard J2EE components:

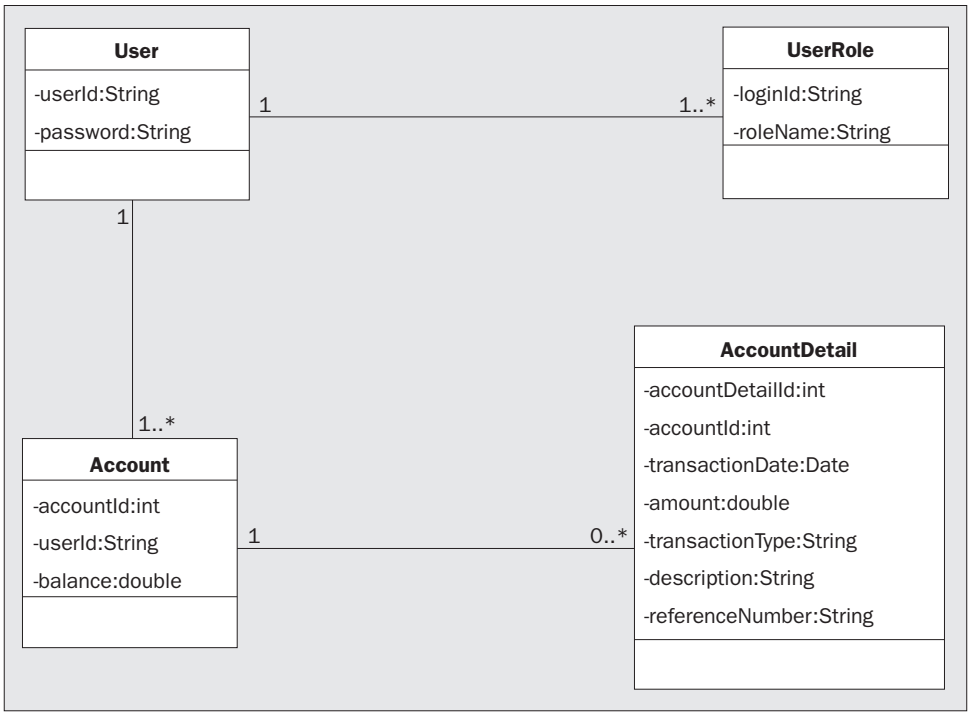
- The web tier will be implemented using a controller servlet, a set of JSP pages, and helper classes
- The business tier will be implemented using Enterprise JavaBean components
- The web tier will interface with the EJB tier using business delegate components

Designing the Database

The domain objects that are identified in the system are:

- `User`
  This represents a user who is authorized to use the system.
- `User Role`
  This entity represents the various roles that are assigned to the user. In this case the roles will represent Regular and Preferred Customers.
- `Account`
  This represents the account details that are held by the users.
- `Account Detail`
  This represents the transaction details associated with each account.

The diagram below depicts the domain model of the system:



The diagram above shows the following details:

- A user has a user ID and password and may have one or more roles and one or more accounts
- An account can inform a user of its closing balance and may have zero or more transactions associated with it
- An account detail has the transaction date, amount involved, transaction type, description, and a reference number

The domain objects will persist in relational database tables. The script below shows the SQL commands for creating the tables and adding some sample data:

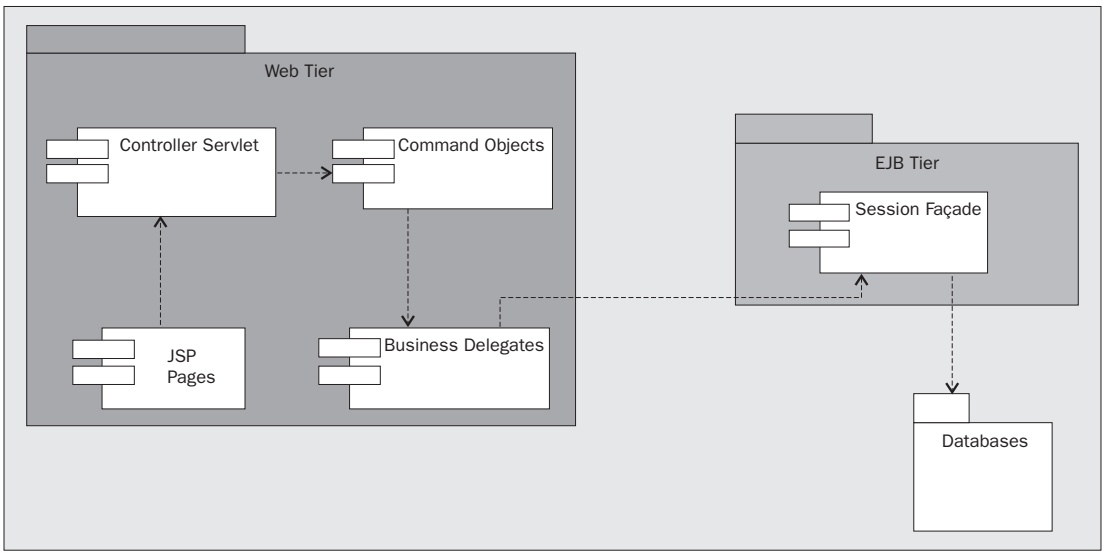
```
DROP TABLE WR_ACCOUNT;
CREATE TABLE WR_ACCOUNT (
  id NUMBER PRIMARY KEY,
  user_id NUMBER,
  balance NUMBER);

DROP TABLE WR_ACCOUNT_DETAIL;
CREATE TABLE WR_ACCOUNT_DETAIL (
  id NUMBER PRIMARY KEY,
  account_id NUMBER,
  transaction_date DATE,
```

```
  amount NUMBER,
  transaction_type VARCHAR(30),
  description VARCHAR(60),
  ref_num VARCHAR(30));

DROP TABLE WR_USER;
CREATE TABLE WR_USER (
  id NUMBER PRIMARY KEY,
  login_id VARCHAR2(30),
  pwd  VARCHAR2(30));

DROP TABLE WR_USER_ROLE;
CREATE TABLE WR_USER_ROLE (
  id NUMBER PRIMARY KEY,
  login_id VARCHAR2(30),
  role_name VARCHAR2(30),
  role_group VARCHAR2(30));

INSERT INTO WR_USER VALUES (1, 'zola', 'striker');
INSERT INTO WR_USER VALUES (2, 'gallas', 'defender');

INSERT INTO WR_USER_ROLE VALUES (1, 'zola', 'standard', 'Roles');
INSERT INTO WR_USER_ROLE VALUES (2, 'gallas', 'preferred', 'Roles');

INSERT INTO WR_ACCOUNT VALUES (1, 1, 1000.00);
INSERT INTO WR_ACCOUNT VALUES (2, 2, 20000.00);

INSERT INTO WR_ACCOUNT_DETAIL VALUES (1, 1, '12-Dec-2000', 1000.00, 'Paid in by
cheque', 'No Description', '012345');
INSERT INTO WR_ACCOUNT_DETAIL VALUES (2, 2, '12-Dec-2000', 20000.00, 'Paid in by
cheque', 'No Description', '123456');
INSERT INTO WR_ACCOUNT_DETAIL VALUES (3, 2, '12-Dec-2000', 1000.00, 'Paid in by
cash', 'No Description', '234567');
INSERT INTO WR_ACCOUNT_DETAIL VALUES (4, 2, '12-Dec-2000', -1000.00, 'Paid out by
cheque', 'No Description', '345678');

COMMIT;
```

Application Architecture

When the users access the system through their browsers, the system will authenticate their security credentials. Once authentication is performed, the system will display the transaction details for Preferred Customers and the account balance for Standard Customers. The Preferred Customers will also be able to view the account balance.

The component diagram shown below depicts the high-level application architecture:



- All the HTTP requests originating from the client browsers will be intercepted by a Controller servlet
- This servlet, based on some information present in the request, will choose a Command object that will process the request
- If processing the request involves utilizing the services provided by the EJB tier, the Command object will access the Session Façade providing the business services using Business Delegates
- The Session Façade returns the data from the relational data store using JDBC
- The Command objects will store this data as request scope beans and inform the Controller servlet about the JSP page that will render the next view
- The JSP pages will extract the data from the request scope beans and display it

Application Security

One of the most important aspects of application development using J2EE is that these applications can rely on the container in which they run for a whole host of system-level services. Among these system-level services defined by the J2EE, EJB, and Servlet specifications are security services. J2EE enables applications to define security policies declaratively using deployment descriptors. The Servlet and EJB APIs also provide features for accessing various aspects related to security. Hence it is possible to implement most of the patterns explained earlier using standard J2EE API and deployment descriptor features.

Single Access Point and Check Point Patterns

In the application all public requests will be mapped to a Controller servlet and access to the Controller servlet will be restricted to authenticated users only.

The application will use form-based login for authenticating users.

- **API Feature:**

  `getRemoteUser()` method on `javax.servlet.http.HttpServletRequest` returns the currently authenticated user.

  `getAuthType()` method on `javax.servlet.http.HttpServletRequest` identifies whether the authentication scheme used is `BASIC`, `FORM`, `DIGEST` or `CLIENT_CERT`.

  `getCallerPrincipal()` method of `javax.ejb.EJBContext` returns the principal associated with the security credentials of the currently executing thread.

  `getRemoteUser()` method on `javax.servlet.http.HttpServletRequest` returns the currently authenticated user.

- **Deployment Descriptor Feature:**

  The `<login-config>` element in the web deployment descriptor is used to specify the login scheme.

  The `<method-permission>` element in the EJB deployment descriptor can be used to restrict access to the EJB methods to authenticated threads only.

The Role Pattern

The Command object that handles the initial login will decide the JSP that will display the next view to the user based on the role of the authenticated user.

All the JSP pages are stored in the `WEB-INF` directory and are not available for direct access from the browser.

- **API-Feature:**

  `isUserInRole()` method on `javax.servlet.http.HttpServletRequest` identifies whether the currently authenticated user belongs to the specified role.

  `isCallerInRole()` method on `javax.ejb.EJBContext` provides the same functionality.

- **Deployment Descriptor Feature:**

  The `<security-constraint>`, `<auth-constraint>`, and `<security-role>` elements in the web deployment descriptor can be used to specify role-based access to URI patterns.

  The `<method-permission>` and `<security-role>` elements in the EJB deployment descriptor can be used to specify role-based access to EJB methods.

  The `<security-role-ref>` and `<role-link>` elements in EJB and web deployment descriptors can be used to map roles defined in the container environment to coded roles in the EJB and web components.

Sessions

We'll be relying on the server-provided features to manage sessions in the application.

- **API-Feature**

  The web components can use `javax.servlet.http.HttpSession` for session-based functionality.

  The EJB components can use stateful session beans for implementing session-based functionality.

- **Deployment Descriptor Feature**

  The `<session-config>` element in the web deployment descriptor can be used for setting maximum inactive intervals for sessions.

In addition to the features explained above, J2EE also provides JAAS (Java Authentication and Authorisation Service) for implementing pluggable security modules. Later in the chapter we will see how we can use the RDBMS-based JAAS module provided by the container provider to implement the security policies defined in the deployment descriptor.

Roles Used in the System

The system basically utilizes the following roles:

- Standard: users with this role can only view the current balance of their accounts
- Preferred: users with this role can view their current balances as well as their transaction statements

Request URIs

The table below depicts the request URIs that are served by the system and the roles required to access those URIs:

URI	Role	Description
`index.jsp`	Standard, Preferred	The first page that is accessed
`home.do`	Standard, Preferred	The URI that serves the home page
`balance.do`	Standard, Preferred	The URI that serves the current balance
`statement.do`	Preferred	The URI that serves the current statement

Web Deployment Descriptor

Now we will look at how the security policies are defined in the web tier using the deployment descriptor:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>

    <display-name>WROX Internet Banking Application</display-name>

    <!-- Define the controller servlet -->
    <servlet>
        <servlet-name>Controller</servlet-name>
        <servlet-class>web.ControllerServlet</servlet-class>
        <!-- Map roles to coded values -->
        <security-role-ref>
            <role-name>PRF</role-name>
            <role-link>preferred</role-link>
        </security-role-ref>
        <security-role-ref>
            <role-name>STD</role-name>
            <role-link>standard</role-link>
        </security-role-ref>
    </servlet>

    <!-- Map all the *.do requests to the controller servlet -->
    <servlet-mapping>
<servlet-name>Controller</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <!-- Set the maximum inactive interval to thirty minutes -->
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <!-- Define the home page -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
```

This security constraint identifies the resources that can be accessed by users belonging to either Standard or Preferred roles:

```
    <security-constraint>
        <display-name>WROX Bank Security</display-name>
        <web-resource-collection>
            <web-resource-name>Standard\Preferred Customers</web-resource-name>
            <description>Resource to get bakance and login</description>
            <url-pattern>/balance.do</url-pattern>
            <url-pattern>/home.do</url-pattern>
            <url-pattern>/index.jsp</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>preferred</role-name>
            <role-name>standard</role-name>
        </auth-constraint>
    </security-constraint>
```

This security constraint identifies the resources that can be accessed only by users belonging to the Preferred role:

```
    <security-constraint>
        <display-name>WROX Bank Security</display-name>
        <web-resource-collection>
            <web-resource-name>Preferred Customers</web-resource-name>
            <description>Resource to get statements</description>
            <url-pattern>/statement.do</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>preferred</role-name>
        </auth-constraint>
    </security-constraint>

    <!-- Define form login configuration -->
<login-config>
        <auth-method>FORM</auth-method>
        <realm-name>WROX Bank Authentication</realm-name>
        <form-login-config>
            <form-login-page>/Login.jsp</form-login-page>
            <form-error-page>/LoginError.jsp</form-error-page>
        </form-login-config>
    </login-config>

    <!-- Define the security roles -->
    <security-role>
        <description>Preferred Customers</description>
        <role-name>preferred</role-name>
    </security-role>
    <security-role>
        <description>Standard Customers</description>
        <role-name>standard</role-name>
    </security-role>

</web-app>
```

EJB Deployment Descriptor

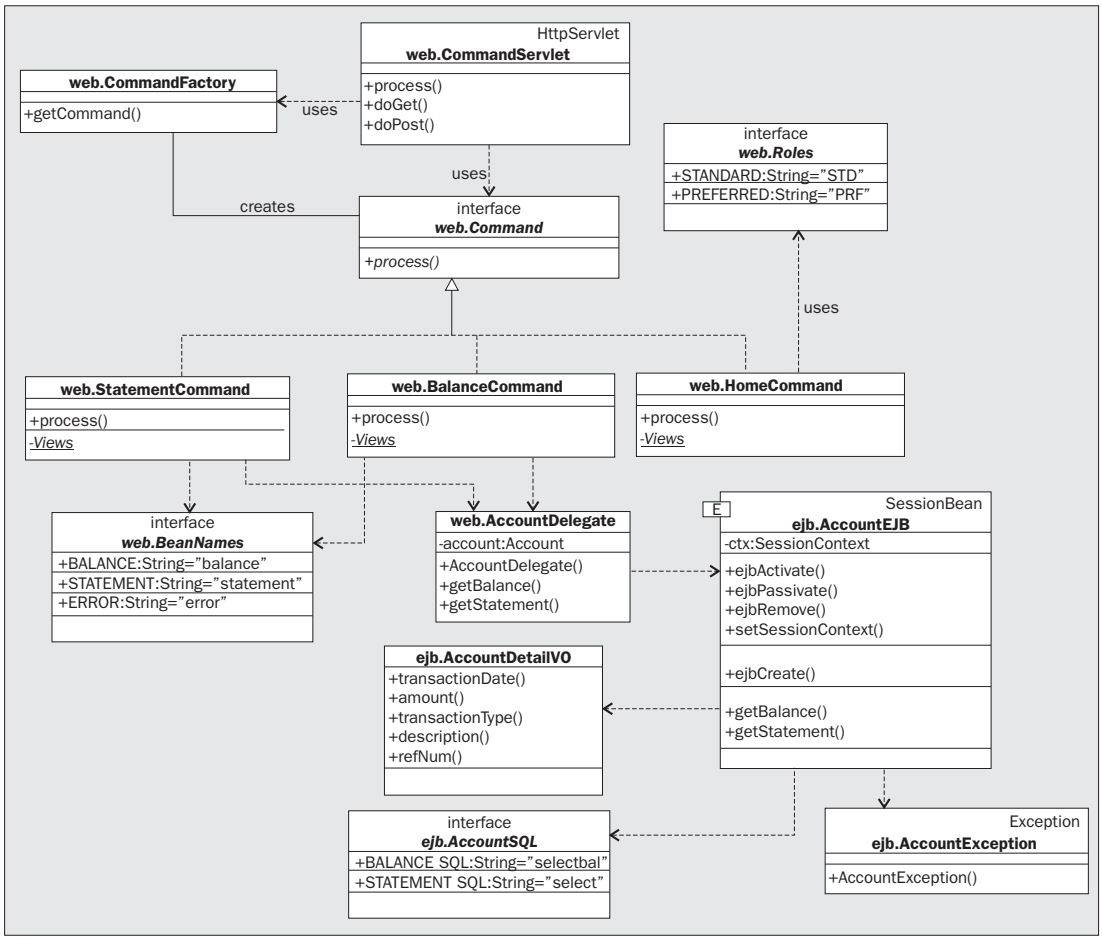
The use cases to get the balance and statement are implemented as methods on a stateless Session Façade. Now we will have a look at how access to these methods is controlled based on roles defined in the EJB deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar>

    <enterprise-beans>

        <session>

            <ejb-name>AccountBean</ejb-name>
```

```
            <home>ejb.AccountHome</home>
            <remote>ejb.Account</remote>
            <ejb-class>ejb.AccountEJB</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>

        </session>

    </enterprise-beans>

    <assembly-descriptor>

        <!-- Define the security roles -->
        <security-role>
            <description>Preferred Customer</description>
            <role-name>preferred</role-name>
        </security-role>
        <security-role>
            <description>Standard Customer</description>
        <role-name>standard</role-name>
        </security-role>

        <method-permission>
            <description>
                Permissions for preferred and standard customers
            </description>
            <role-name>preferred</role-name>
            <role-name>standard</role-name>
            <method>
                <ejb-name>AccountBean</ejb-name>
                <method-name>create</method-name>
            </method>
            <method>
                <ejb-name>AccountBean</ejb-name>
                <method-name>getBalance</method-name>
            </method>
        </method-permission>

        <method-permission>
            <description>Permissions for preferred customers</description>
            <role-name>preferred</role-name>
            <method>
                <ejb-name>AccountBean</ejb-name>
                <method-name>getStatement</method-name>
            </method>
        </method-permission>

    </assembly-descriptor>

</ejb-jar>
```

Application Classes

In this section, we will have a look at the various classes and interfaces used in the application:



The ControllerServlet Servlet

This servlet acts as the front-end controller:

```
package web;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;

import java.io.IOException;
```

```
public class ControllerServlet extends HttpServlet {

    public void process(HttpServletRequest req, HttpServletResponse res)
           throws ServletException, IOException {

        try {
```

Get the command from the factory based on the servlet path:

```
            Command com = CommandFactory.getCommand(req.getServletPath());
```

Get the command to process the request and retrieve the JSP page for the next view:

```
            String view = com.process(req, res);
```

Forward the request to the JSP page using request dispatcher:

```
            RequestDispatcher rd = getServletContext().getRequestDispatcher(view);
            rd.forward(req, res);

        } catch(Throwable th) {
            throw new ServletException(th);
        }
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
          throws ServletException, IOException {
        process(req, res);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {
        process(req, res);
    }
```

The Command Interface

This interface defines the contract for all the Command objects used in the system:

```
package web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface Command {
```

This method should be implemented by all the Command objects:

```
    public String process(HttpServletRequest req, HttpServletResponse res);

}
```

The Roles Interface

This interface enumerates the coded role values used in the web tier:

```
package web;

public interface Roles {
```

The coded role used in the web tier for Standard users:

```
    public static final String STANDARD = "STD";
```

The coded role used in the web tier for Preferred users:

```
    public static final String PREFERRED = "PRF";
}
```

The BeanNames Interface

This interface enumerates the bean names used in the JSP pages:

```
package web;

public interface BeanNames {
```

Bean name under which the balance is stored:

```
    public static final String BALANCE = "balance";
```

Bean name under which the statement is stored:

```
    public static final String STATEMENT = "statement";
```

Bean name under which the error message is stored:

```
    public static final String ERROR = "error";
}
```

The StatementCommand Command Object

This Command object handles the statement requests:

```
package web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import ejb.AccountException;

public class StatementCommand implements Command {
```

The inner interface that defines the possible views for the command:

```
    private static interface Views {

        public static final String STATEMENT = "/WEB-INF/Statement.jsp";
        public static final String ERROR = "/WEB-INF/Error.jsp";

    }

    public String process(HttpServletRequest req, HttpServletResponse res) {

        try {
```

Use the account delegate to get the balance and store it as a request scope bean and return the name as the JSP page that will display the balance:

```
            AccountDelegate delegate = new AccountDelegate();
            req.setAttribute(BeanNames.STATEMENT, delegate.getStatement());
            return Views.STATEMENT;

        } catch(AccountException ex) {
```

If an error occurs, store the error message as a request scope bean and return the error JSP name:

```
            req.setAttribute(BeanNames.ERROR, ex.getMessage());
            return Views.ERROR;
        }
    }
}
```

The BalanceCommand Command Object

This Command object handles the balance requests:

```
package web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import ejb.AccountException;

public class BalanceCommand implements Command {
```

The inner interface that defines the possible views for the command:

```
    private static interface Views {

        public static final String BALANCE = "/WEB-INF/Balance.jsp";
```

```
        public static final String ERROR = "/WEB-INF/Error.jsp";

    }
    public String process(HttpServletRequest req, HttpServletResponse res) {

        try {
```

Use the account delegate to get the statement and store it as a request scope bean and return the name of the JSP page that will display the statement:

```
            AccountDelegate delegate = new AccountDelegate();
            req.setAttribute(BeanNames.BALANCE, delegate.getBalance());
            return Views.BALANCE;

        } catch(AccountException ex) {
```

If an error occurs, store the error message as a request scope bean and return the error JSP page name:

```
            req.setAttribute(BeanNames.ERROR, ex.getMessage());
            return Views.ERROR;
        }
    }
}
```

The HomeCommand Command Object

This Command object decides which page to display after the initial login based on the role of the authenticated user:

```
package web;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HomeCommand implements Command {
```

The inner interface that defines the possible views for the command:

```
    private static interface Views {

        public static final String STATEMENT = "/statement.do";
        public static final String BALANCE = "/balance.do";

    }

    public String process(HttpServletRequest req, HttpServletResponse res) {
```

Return the URI to display statement for preferred customers:

```
            if(req.isUserInRole(Roles.PREFERRED)) return Views.STATEMENT;
```

Return the URI to display balance for standard customers:

```
            else if(req.isUserInRole(Roles.STANDARD)) return Views.BALANCE;
            else throw new IllegalArgumentException("Unexpected Role.");
        }
    }
```

The CommandFactory Factory

This class implements a factory based approach to create Command objects:

```
package web;

public class CommandFactory {

    public static Command getCommand(String path) {
```

Return a Command object based on the specified servlet path:

```
            if(path.equals("/home.do")) return new HomeCommand();
            else if(path.equals("/balance.do")) return new BalanceCommand();
            else if(path.equals("/statement.do")) return new StatementCommand();
            else throw new IllegalArgumentException("Invalid path:" + path);
        }

    }
```

The Account Remote Interface

This is the component interface for the Session Façade that implements the use cases:

```
package ejb;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

import java.util.ArrayList;

public interface Account extends EJBObject {
```

Business method to get the balance:

```
    public Double getBalance() throws RemoteException, AccountException;
```

Business method to get the statement:

```
    public ArrayList getStatement() throws RemoteException, AccountException;

}
```

The AccountHome Home Interface

```
package ejb;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;

public interface AccountHome extends EJBHome {

    public Account create() throws RemoteException, CreateException;

}
```

The AccountEJB Bean Class

This is the bean class for the Session Façade that implements the use cases:

```
package ejb;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.ejb.SessionBean;
import javax.ejb.EJBException;
import javax.ejb.SessionContext;

import java.util.ArrayList;

public class AccountEJB implements SessionBean {

    private SessionContext ctx;

    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
```

```
    public Double getBalance() throws AccountException {

        Connection con = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        InitialContext iCtx = null;
        DataSource ds = null;

        try {
```

Get the login ID of the currently authenticated user. The security context is propagated to the EJB container from the web container:

```
            String loginId = ctx.getCallerPrincipal().getName();
```

Look up the datasource:

```
            iCtx = new InitialContext();
            ds = (DataSource)iCtx.lookup("java:/AccountDS");
```

Issue the SQL to get the balance for the authenticated user:

```
            con = ds.getConnection();
            stmt = con.prepareCall(AccountSQL.BALANCE_SQL);
            stmt.setString(1, loginId);

            rs = stmt.executeQuery();
```

Return the balance:

```
            if(rs.next())
                return new Double(rs.getDouble(1));

            throw new AccountException("Account not found.");

        } catch(NamingException ex) {
            throw new EJBException(ex);
        } catch(SQLException ex) {
            throw new EJBException(ex);
        } finally {

            try {
                if(iCtx != null) iCtx.close();
                if(rs != null) rs.close();
                if(stmt != null) stmt.close();
                if(con != null) con.close();
            } catch(Throwable th) {
                th.printStackTrace();
            }
        }
    }
```

```
    public ArrayList getStatement() throws AccountException {

        Connection con = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        InitialContext iCtx = null;
        DataSource ds = null;

        try {
            String loginId = ctx.getCallerPrincipal().getName();

            iCtx = new InitialContext();
            ds = (DataSource)iCtx.lookup("java:/AccountDS");
```

Issue the SQL to get the statement for the authenticated user:

```
            con = ds.getConnection();
            stmt = con.prepareCall(AccountSQL.STATEMENT_SQL);
            stmt.setString(1, loginId);

            rs = stmt.executeQuery();

            ArrayList statement = new ArrayList();
            AccountDetailVO vo;
            while(rs.next()) {
                vo = new AccountDetailVO();
                vo.transactionDate = rs.getTimestamp(1);
                vo.amount = rs.getDouble(2);
                vo.transactionType = rs.getString(3);
                vo.description = rs.getString(4);
                vo.refNum = rs.getString(5);

                statement.add(vo);
            }
```

Return the statement:

```
            if(statement.size() > 0)
                return statement;

            throw new AccountException("Account not found.");

        } catch(NamingException ex) {
            throw new EJBException(ex);
        } catch(SQLException ex) {
            throw new EJBException(ex);
        } finally {
            try {
                if(iCtx != null) iCtx.close();
                if(rs != null) rs.close();
                if(stmt != null) stmt.close();
                if(con != null) con.close();
```

```
            } catch(Throwable th) {
                th.printStackTrace();
            }
        }
    }
}
```

The AccountSQL Interface

This interface enumerates the SQL commands used in the system:

```
package ejb;

public interface AccountSQL {
```

SQL to get the balance:

```
    public static final String BALANCE_SQL =
      "SELECT balance " +
      "FROM WR_ACCOUNT, WR_USER " +
      "WHERE WR_ACCOUNT.user_id = WR_USER.id " +
      "AND WR_USER.login_id = ?";
```

SQL to get the statement:

```
    public static final String STATEMENT_SQL =
      "SELECT transaction_date, amount, transaction_type, description, ref_num " +
      "FROM WR_ACCOUNT, WR_USER, WR_ACCOUNT_DETAIL " +
      "WHERE WR_ACCOUNT.user_id = WR_USER.id " +
      "AND WR_ACCOUNT.id = WR_ACCOUNT_DETAIL.account_id " +
      "AND WR_USER.login_id = ?";

}
```

The AccountDetailVO Value Object

This is the value object used to transfer account detail data from the EJB tier to the presentation tier:

```
package ejb;

import java.sql.Timestamp;

public class AccountDetailVO {

    public Timestamp transactionDate;
    public double amount;
    public String transactionType;
    public String description;
    public String refNum;

}
```

The AccountException Exception

This is the business exception used by the Session Façade:

```
package ejb;

public class AccountException extends Exception {

    /* Creates a new instance of AccountException */
    public AccountException() {
    }

    public AccountException(String message) { super(message); }

}
```

The AccountDelegate Delegate Class

```
package web;

import ejb.Account;
import ejb.AccountHome;
import ejb.AccountException;

import javax.rmi.PortableRemoteObject;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.rmi.RemoteException;

import javax.ejb.CreateException;

import java.util.ArrayList;

public class AccountDelegate {
```

A reference to the Session Façade:

```
    private Account account;

    /* Creates a new instance of AccountDelegate */
    public AccountDelegate() {

        try {
```

Look up the home reference for the Session Façade and use it to create the remote reference:

```
            InitialContext ctx = new InitialContext();
            Object obj = ctx.lookup("AccountBean");
            AccountHome home =
               (AccountHome)PortableRemoteObject.narrow(obj, AccountHome.class);
```

```
            account = home.create();

        } catch(NamingException ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        } catch(RemoteException ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        } catch(CreateException ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        }

    }

    public Double getBalance() throws AccountException {

        try {
```

Delegate the method call to the Session Façade:

```
            return account.getBalance();

        } catch(AccountException ex) {
            throw ex;
        } catch(Throwable ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        }
    }
    public ArrayList getStatement() throws AccountException {

        try {
```

Delegate the method call to the Session Façade:

```
            return account.getStatement();

        } catch(AccountException ex) {
            throw ex;
         }catch(Throwable ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex.getMessage());
        }
    }
}
```

JSP Pages Used in the Application

Now, we will have a look at the various JSP pages that are used in the application:

The index.jsp Page

This is the welcome JSP page. This simply forwards the request to the URI `home.do`:

```
<%@page contentType="text/html"%>
 <jsp:forward page="/home.do"/>
```

The Login.jsp Page

This JSP page is used by the container to perform authentication when an unauthenticated user tries to access the system:

```
<html>
    <head>
        <title>Login Page for Examples</title>
    </head>
    <body bgcolor="white">
        <form method="POST" action='<%= response.encodeURL("j_security_check") %>'
        >
            <table border="0" cellspacing="5">
                <tr>
                    <th align="right">Username:</th>
                    <td align="left"><input type="text" name="j_username"></td>
                </tr>
                <tr>
                    <th align="right">Password:</th>
                    <td align="left"><input type="password"
                    name="j_password"></td>
                </tr>
                <tr>
                    <td align="right"><input type="submit" value="Log In"></td>
                    <td align="left"><input type="reset"></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

The LoginError.jsp Page

This JSP page is used by the container if authentication fails:

```
<html>
    <head>
        <title>Error Page For Examples</title>
    </head>
    <body bgcolor="white">
        Invalid username and/or password,
    </body>
</html>
```

The Statement.jsp Page

This JSP is used for displaying the statement:

```
<%@page contentType="text/html" import="java.util.*,ejb.AccountDetailVO"%>
<html>
    <head>
        <title>Account Statement</title>
    </head>
    <body>

        <jsp:useBean id="statement" scope="request" type="java.util.ArrayList" />
        <table>
            <tr>
                <th>Date</th>
                <th>Amount</th>
                <th>Type</th>
                <th>Description</th>
                <th>Reference</th>
            </tr>
            <%
                Iterator it = statement.iterator();
                while(it.hasNext()) {
                    AccountDetailVO vo = (AccountDetailVO)it.next();
            %>
            <tr>
                <td><%= vo.transactionDate %></td>
                <td><%= vo.amount %></td>
                <td><%= vo.transactionType %></td>
                <td><%= vo.description %></td>
                <td><%= vo.refNum %></td>
            </tr>
            <%
                }
            %>
        </table>

        <a href="balance.do">View Balance</a>
    </body>
</html>
```

The Error.jsp Page

This JSP page is used to display error messages:

```
<%@page contentType="text/html" isErrorPage="true" import="web.BeanNames" %>
<html>
<head><title>JSP Page</title></head>
<body>

    <%= request.getAttribute(BeanNames.ERROR) %>

</body>
</html>
```

Summary

In this chapter, we have seen the following patterns associated with security:

- The Single Access Point pattern
- The Check Point pattern
- The Role pattern

We have also implemented these patterns in the case study Wrox Web Bank Application using the standard J2EE API and deployment descriptor features. These patterns can be applied to the design and development of applications, to configuration and management of hosts, and so on; however, these patterns do not define specific technologies nor coding styles. The ultimate goal of these patterns is to support not only the application security but also the host and network security.