Chapter 12

Web Application: Vacation Tracking System

For many businesses today, the independence of workers has been ever increasing. It is not uncommon for workers to divide their time across multiple projects and to report to multiple project managers. As a result, managers have fewer informal interactions with their workers and find it increasingly difficult to be aware of and manage their workers' vacation time. Thus, in the example explored in this chapter, our fictitious company has decided to develop and deploy a flexible vacation time management application for managers and employees alike to use to manage their vacation time.

The decision by a large enterprise organization to implement this and similar functionality is never made in isolation. It is unlikely that this proposed system is the first that this organization has ever created. This desired functionality must be considered in the context of any existing systems (and perhaps even with other proposed systems). As a result, many architectural decisions will already be made a priori, and in our situation, the decision to deliver this functionality as a Web application not only is well suited to the task but also will be considered an extension to the organization's existing intranet, thus providing a convenient and natural entry point for its users.

In the following sections, we discuss a summary of the architecturally significant and interesting aspects of this system. Given the space limitations of this book, we express only those aspects and content necessary to impart a general feel and understanding of the application. In the real world, an accurate and complete discussion of such a system would require significantly more content, models, and examples. However, here we cover the key or central principles of object orientation as they apply to this sort of Web application development, and we discuss several interesting issues unique to the Web application development process.

12.1 Inception

The system's requirements are presented by a summary of the vision document, key features, the use case model, key use case specifications, and architecturally significant line item requirements.

The Requirements

The vision for this project can be summarized easily.

> A Vacation Tracking System (VTS) will provide individual employees with the capability to manage their own vacation time, sick leave, and personal time off, without having to be an expert in company policy or the local facility's leave policies.

The most important goal of this system is to give individual employees the capability and responsibility to manage this particular aspect of their employment agreements with the company. The underlying motivations for this desire include the need to streamline the functions of the human resources (HR) department, to minimize noncore, business-related activities of management, and to give a sense of empowerment to the employees. These objectives will be met only if the system developed is easy to use, intuitive, and intelligent. An overriding design goal can therefore be stated simply.

> The system must be easy to use.

Anyone reading this with even the most minimal experience developing commercial software as part of a team effort will be rolling their eyes at such a vague requirement. Something so blatantly general and subjective should never be recorded as an official requirement, right? Not necessarily so. While it is clear that this simple line item is not a hard and traceable requirement, it does represent an honest feature of the desired system. It is so important that unless it at least appears to have been met by the developers and ultimately accepted by the end users alike, the delivered application will fail.

High-level and potentially vague features like this sometimes need to be part of the official requirements set, not so they can be objectively verified and tracked through testing, but rather so they can be used to support and justify other, more concrete requirements or design decisions. As an example, consider the requirements for a Web application that at some point requires the end user to submit to the system some formatted text (e.g., bold, italics, lists, paragraphs, and so on). The architects have two potential solutions. They can require the end user to incorporate special codes into the body of the text, as many popular bulletin board systems do today, or they can use a custom-built or commercially available Java applet that provides WYSIWYG formatting. The technical complexity and risk of such an applet are significant, yet the end-user advantages are also equally clear. If one of the primary system features or design goals was ease of use, the use of this applet could be justified. However, if ease of use was simply a desired rather than critical requirement, introducing the potential complexity and risk of the applet could not be justified.

The main goal of this application is to improve the internal business processes of this organization, at least with respect to the time it takes to manage vacation time requests. In the past, all vacation time had to be approved by an immediate manager and then checked by a clerk in the HR department before it was authorized. Sometimes this manual process could take days. An automated system will speed up this process and will require at most one manual approval by the immediate manager (some high-level employees may not require manager approval).

This system has the potential to save time and money mostly in the HR department, which is essentially taken out of the individual time request process and replaced by a rules-based validation system. HR personnel are still responsible for entering and updating employee vacation data in the system; however, they will no longer be a link in the chain for requesting and validating each time request.

The system will provide the following key features:

- Implements a flexible rules-based system for validating and verifying leave time requests
- Enables manager approval (optional)
- Provides access to requests for the previous calendar year, and allows requests to be made up to a year and a half in the future
- Uses e-mail notification to request manager approval and notify employees of request status changes
- Uses existing hardware and middleware
- Is implemented as an extension to the existing intranet portal system, and uses the portal's single-sign-on mechanisms for all authentication
- Keeps activity logs for all transactions
- Enables the HR and system administration personnel to override all actions restricted by rules, with logging of those overrides
- Allows managers to directly award personal leave time (with system-set limits)

- Provides a Web service interface for other internal systems to query any given employee's vacation request summary
- Interfaces with the HR department legacy systems to retrieve required employee information and changes

The Use Case Model

The top-level use case model contains four actors and eight use cases, as shown in Figure 12–1. The granularity of the use cases is reasonably coarse. For example, the use case `Manage Time` describes functionality, invoked by the `Employee`, that includes viewing, creating, and canceling vacation time requests. A use case is not a description of a single functional requirement but rather a description of something that provides significant value to the actor invoking it, in the form of scenarios. Just being able to view a vacation time request, for example, provides minimal value, but being able to manage your own vacation time does provide significant value.

An interesting observation about use cases of Web-centric systems is that they tend to be expressed very strictly in terms of stimulus and response. That is, a use case scenario is typically expressed as a list of actor actions and immediate system

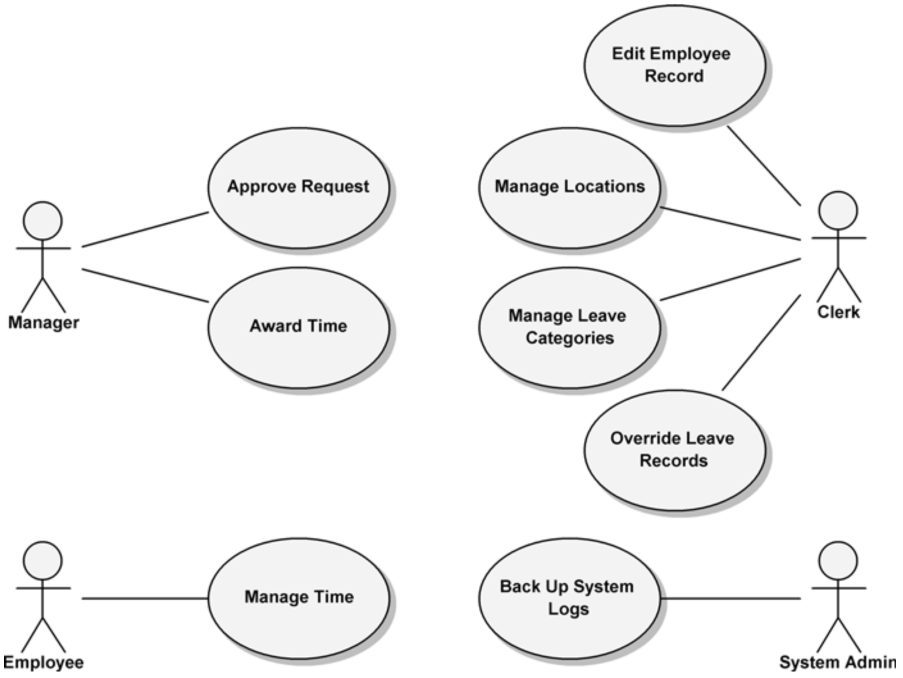

Figure 12–1 The Top-Level Use Case Model

responses. Also, there is typically a high degree of correlation of system responses to actual and individually named Web pages or screens. This is in contrast to non-Web applications, where content in the scenarios of activity focuses more on the information and use gestures being exchanged than on the identification of discrete user interface units, such as Web pages in a Web-centric system.

The system contains the following actors.

- `Employee`: The main user of this system. An employee uses this system to manage his or her vacation time.
- `Manager`: An employee who has all the abilities and goals of a regular employee, but with the added responsibility of approving vacation requests for immediate subordinates. A manager may award subordinates comp time, subject to certain limits set in the system.
- `Clerk`: A member of the HR department who has sufficient rights to view employees' personal data and is responsible for ensuring that employees' information in all HR systems is up to date and correct. An HR clerk can add or remove nearly any record in the system. In the real world, HR clerks may or may not be employees; however, if they are employees, they use two separate login IDs to manage these two different roles.
- `System Admin`: A role responsible for the smooth running of the system's technical resources (e.g., Web server, database) and for collecting and archiving all log files.

The main use cases are as follows.

- `Manage Time`: Describes how employees request and view vacation time requests.
- `Approve Request`: Describes how a manager responds to a subordinate's request for vacation time.
- `Award Time`: Describes how a manager can award a subordinate extra leave time (comp time).
- `Edit Employee Record`: Describes how an HR clerk edits an employee's information in the system. This includes setting all the leave time allowances and the maximum time that can be awarded by the manager.
- `Manage Locations`: Describes how an HR clerk manages location records and their rules.
- `Manage Leave Categories`: Describes how an HR clerk manages leave categories and their rules.
- `Override Leave Records`: Describes how an HR clerk may override any rejection of leave time requests made by the rules in the system.
- `Back Up System Logs`: Describes how the system administrator backs up the system's logs.

12.2 Elaboration

Sometimes it is not always clear when analysis starts or when requirements gathering and understanding during the Inception phase end. This is also why iterative development processes are so popular and the practicality of the waterfall process so often questioned. It is important, however, to have the most important and architecturally significant use cases described and discussed first. All the details need not be complete, but the architecturally significant ones should be addressed before a particular use case can undergo refinement.

In the ideal world, analysis of a system should be independent of an implementing architecture. The reality, however, is that prior knowledge of the implementing architecture may contribute to the shape of the analysis.

This can be especially true when the application under development is a Web application because in most cases the decision to adopt a Web-centric architecture is something that is known at Inception. In our case here, the idea that the solution will ultimately be delivered as a Web-centric application appears in the detailed requirements. For example, the statement in the main flow of the `Manage Time` use case (discussed later), "The employee should have access to a visual calendar to help select and compare selected dates," is prompted in part by the knowledge that most Web browsers do not have a common date picker or calendar widget. In the specification of a native Windows client application, this assumption of such a capability might not have been explicitly brought out since such controls are in common use on Windows-based native client applications. The use case writer in this case is explicitly identifying an area where the system needs to be user friendly. A more subtle and pervasive indication that knowledge of the architecture is known during the process of use case writing is the constant reference to "navigating to Web pages" and "submitting" information, concepts linked tightly with Web-centric architectures.

Sadly, there is no commonly agreed-upon way to quantitatively represent an arbitrary architecture. Here, we will use the 4+1 architecture view model [3], as described in Chapter 6.

The following brief descriptions of a Web application's architecture are not meant to be a complete discussion of Web-centric architectures; however, we do cover enough significant aspects here to help explain design decisions made later.

The Deployment View

A Web application, being a specialization of a client/server application, has minimally two main nodes, the server and the client browser. The server is a node that has a known address on a network and is configured to listen for HTTP requests on a specific port, typically port 80. A client browser application makes a request, at the behest of the user, for an HTML-formatted resource on the server. The server, most likely, will be concurrently running a number of services, including other Web applications, possibly a database server, an application server, and so on. In Figure 12–2, the `Client` and `Server` nodes are clearly identified.

In the Deployment View of the key components, execution environments `Tomcat` and `Cloudscape` are treated as nested nodes of the server. The `Tomcat` node is a Web application execution environment based on the Java environment. The `Tomcat` node itself is shown here deploying the artifact `VTSWeb.war`, a Web application archive file. The `Cloudscape` execution environment is a database server capable of executing SQL files and shown here deploying an artifact called `VTS.sql`.

The `Client` node in Figure 12–2 is shown deploying the `Firefox.exe` artifact, which is an executable HTML browser application. The `Client` node has a communication link to the `Server` node. This communication link may be anything from a dialup ISP to broadband or wireless. The important and logical aspect to concentrate on is that the client communicates to the nested Web and database servers via this communication link.

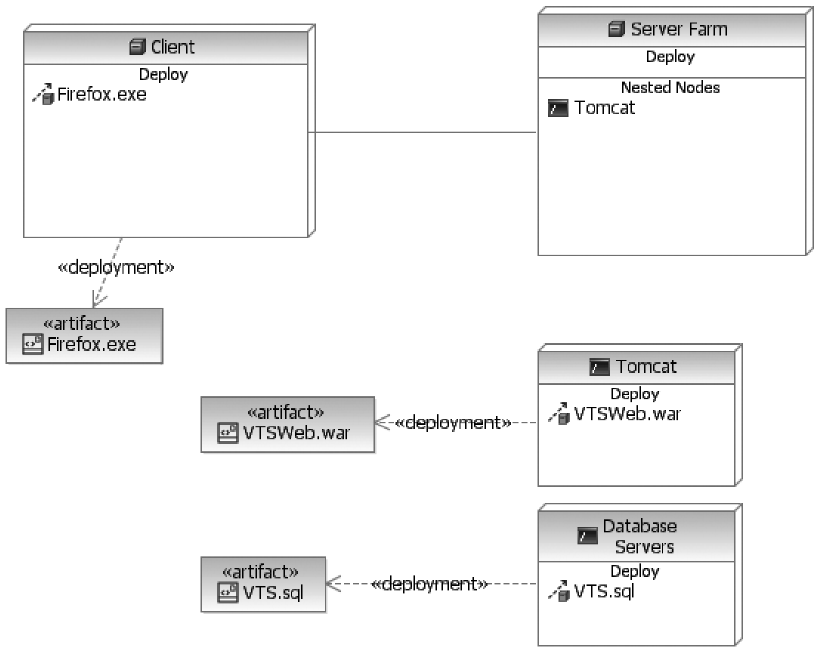

Figure 12–2 The Deployment View of the Components in a Web-Centric System

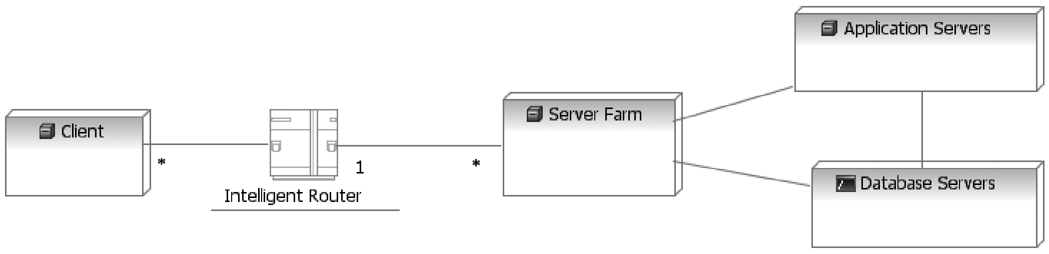

Figure 12–3 A Deployment Diagram for a Web Application Server Farm with Parallel and Redundant Processing Nodes

This representation of a Web-centric system is simplistic. In larger systems, the deployment diagrams showing the topography of all the nodes, components, servers, and communication links can be complex, resulting in the need for a good visualization and documentation system such as the UML. Figure 12–3 is an abstract deployment diagram showing how an intelligent router can be used with a Web server farm and how an entire set of applications can be run from application servers. The details of such strategies can be quite complex, but the general strategy of scaling an application by adding additional processing nodes is certainly viable in Web applications.

The Logical View

The typical Web application has at least four logical components: the browser running on the client, the Web or application container, a separate component for the application logic itself, and a database server component. In Figure 12–4, the `Firefox.exe` component is a commonly available multiplatform browser, and `Tomcat` is a popular Web container based on the Java Server Pages (JSP) specification. The `VTSWeb` component represents the business application, and `Cloudscape` is a simple portable database server. The most important logical point of this diagram is that the client browser is never in direct contact with the database or even the business application component. All access to server-side resources is mediated by the Web container. This makes the Web container an important component to consider when thinking about security requirements.

Web servers are designed to listen to certain ports, usually port 80, and respond to `GET` and `POST` requests. `GET` and `POST` are commands defined in the HTTP protocol. They are essentially two simple ways to request information from a Web server. The `POST` method is a little more extensible and used more often when data or files supplied by the user are sent to the server. The determination of which command to use is embedded in the HTML-formatted page being rendered by the browser. Usually the information returned by a browser request is an HTML-formatted document, which can be visually rendered in the browser, but

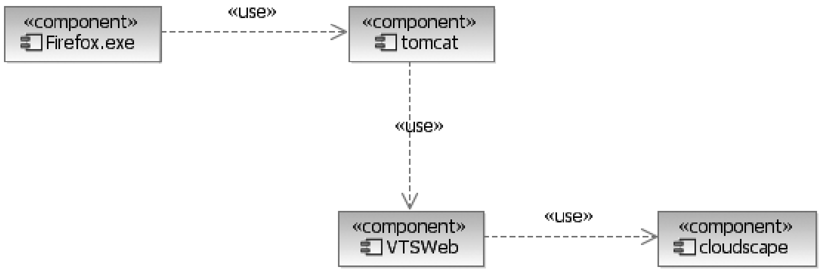

Figure 12–4 The Logical View of the Primary Components of a Web Application

the server could return streamed data in any form, leaving the client responsible to save or delegate the processing of the information to another locally installed application.

Figure 12–5 shows logical classes representing elements in both the client and server tiers of the application. On the client, a browser is responsible for requesting Web pages (or more generically, resources) with simple HTTP `GET` or `POST` commands. These services can be provided by the operating system or may be implemented in the browser itself (in which case the browser calls on lower-level network APIs). The Web container is constantly listening on certain ports (e.g., port 80) for incoming HTTP requests. An HTTP request is packaged by the browser and sent to the Web container. It contains a resource identifier that points to a specific Web page or references a Web application. The request can have accompanying it a set of key-value pairs (parameters), all represented as strings.[1] Some HTTP requests are packaged with more complex form data that the user supplies just before the page request is submitted.

When the Web container receives a request for a resource, it must first determine the file or application resource to invoke. The container invokes the proper resource, and if the resource indicates that it is dynamic and should be processed, the Web container executes it. During processing, the HTTP request information (parameters and form data) is examined, and the application performs the necessary business logic. Often this logic is executed in a separate application server (e.g., an EJB container), which potentially accesses another database server.

Logically, a Web application is made up of Web pages, controllers, and entities (the three basic stereotypes found in an analysis model). Static Web pages (no

1. The gory details of parameter and post data formatting can be found at the World Wide Web Consortium Web site (www.w3.org). Fortunately, most of these details are managed by the Web application framework on which we choose to implement the application.

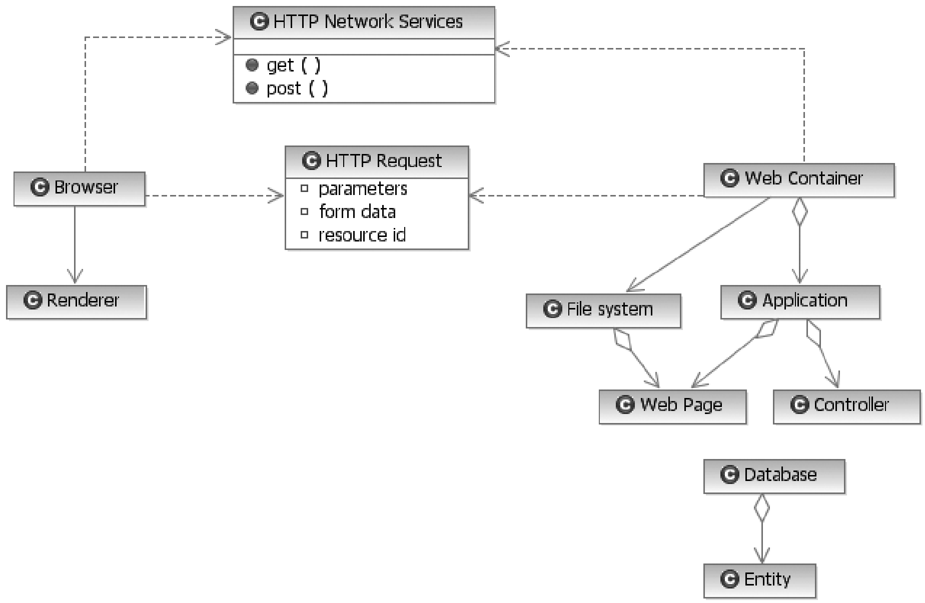

Figure 12–5 The High-Level Logical View of the Objects Involved in a Web Application

processing) may just reside in a file system, while Web pages that contain business logic processing must be loaded and executed in the context of a container. Controller objects are often embedded in components, and persistent entities are managed by databases.

The Process View

There are minimally two, but usually more, processes involved in a typical Web application. The client and server operate asynchronously, except during the handling of HTTP requests. Additional database, authentication, and messaging servers are often part of a typical Web application. They may coexist on a single server node or be distributed across multiple nodes. The process layout of a Web application has the same flexibility as any client/server architecture, with only the one requirement that client nodes must run some form of Web browser client software to initiate communication with the server-side application.

As intimated earlier, the most important thing to understand about Web application architectures is that they work in a connectionless mode. That is, the client and server are never connected longer than it takes to process the one `GET` or `POST` request. Once a server resource (i.e., HTML-formatted Web page) is requested by a browser and the server responds with that resource, the connection

Client State Management

One of the most interesting problems that Web applications face is that of managing client state on the server in a connectionless environment. Since every request and response made between client and server is completed with a new and unique connection, it is difficult for a server to keep track of the sequence of requests of any one particular client.

Managing state is important for many applications since a single use case scenario often involves navigating through a number of different Web pages. If there were no state management mechanism, you would have to continually supply all previous information entered for each new Web page. Even for the simplest applications this can get tedious. Imagine having to reenter the contents of your shopping cart from scratch every time you visit it, or entering your user name and password for every screen you visit while checking your Web-based e-mail.

The most commonly implemented solution to this problem, originally proposed by the World Wide Web Consortium (W3C), is the HTTP State Management Mechanism or, as it is more popularly known, cookies. A cookie is a piece of data that a Web server can ask a Web browser to hold on to and to return every time the browser makes a subsequent request for a HTTP resource from that server. Typically, the size of the data is small, between 100 and 1000 bytes. Web application frameworks manage client state by generating a unique identifier each time a browser first interacts with the application and places this ID in a cookie for the browser. This ID is used as a key into a map on the server that contains all the state information for that particular client.

URL redirection is an alternative approach to cookies for managing client state. In this approach, every hyperlink and form places the ID on the end of the URL submitted to the server during the next page request. The ID performs the same role as a cookie, but as a parameter of a URL it can be used with browsers that have explicitly turned off their cookie feature. The downside to this approach is that every page in a Web site must be dynamic, and the user cannot wander outside of the application in the middle of a use case scenario, lest the particular session between the client and server be broken.

The two other approaches for managing client state serialize the entire state into the URL parameters or into cookies. These are not commonly used for a number of reasons, the least of which is that they are inherently less secure since the client's state is sent over the wire back to the browser, whereas in the previous approaches the state is managed solely on the server. Also, when all state values are placed in cookies, they are limited by size (4K) and can have at most 20 cookies per domain. All state data must be encoded into simple text (no white space, semicolons, and so on). This also implies that you can't easily capture client state with higher-level objects in the session state.

between client and server is broken. Figure 12–6 shows a client making a simple GET request to the server (Tomcat). The server determines from the request which Web application and resource to invoke. A Web page inside the application is instantiated or invoked, and this represents the main trigger for the execution of business logic. Nearly all business logic in a Web application is invoked during the process of handling a GET or POST request. When the logic is finished, the application is responsible for preparing a response page (i.e., the next Web page in the scenario). It is important to realize that once the request for a Web resource is fulfilled, the application stops working for that particular client.

The implications are that it is not at all obvious how a server application can keep track of one particular client's request history and hence the relative internal state of the client. For example, without exploiting certain features of HTTP or implementing certain architectural mechanisms, a server application would have difficulty managing even a simple shopping cart or the state of a particular user walking through a multistepped wizard. Fortunately, most Web application environments provide many useful utilities and mechanisms for managing client state.

Another implication of this architecture is that the server does not know whether the user has abandoned the application in the middle of some business process. It is entirely likely that on occasion a remote user might become disconnected from the network and hence be unable to finish a particular business process that was started. In a more classic client/server application, the server might receive a notification that the client has been prematurely disconnected, but in a Web application, there are no notifications sent to the server when a user becomes disconnected or simply decides to just shut down the browser.

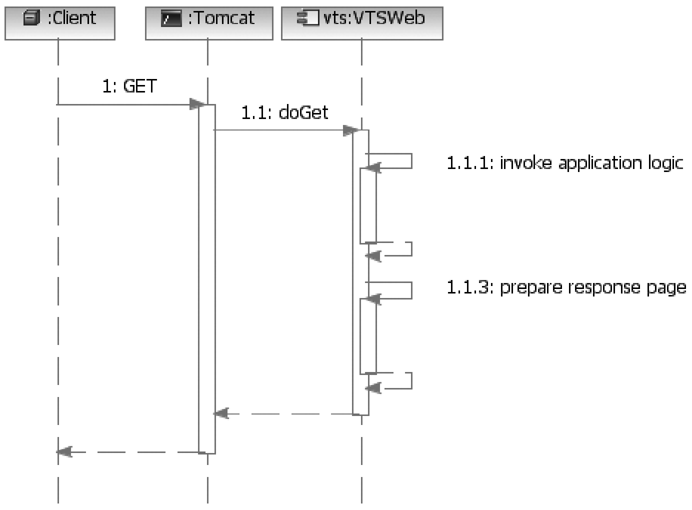

Figure 12–6 A Basic Sequence Diagram of an HTTP GET Request

Web application designs therefore must be very mindful of what resources are opened and accessed between Web page requests. For example, one cardinal rule of Web application design is to never open a transaction in one page and close it in another. The time between Web page requests from a single client is usually on the order of seconds and could at any time abruptly stop. Managing transactions and locks on this order of magnitude is surely going to bring an application server to its knees.

The Implementation View

Figure 12–7 shows an overview of the implementation model for the Vacation Tracking System and describes the system's tiers and packages. In this diagram, the main Web tier component, `VTSWeb.war`, is shown containing Web page artifacts (JSP and HTML files) as well as a set of Java classes in the `web` and `sdo` packages. This code is used to process and invoke the EJB logic in the business tier. The package `com.acme.vts` is shown, in the background, as the main namespace for all the Java components. The Deployment View would describe in more detail the deployment of the components in the tiers, and the Logical View would describe in more detail the nature and responsibility of the components.

The Use Case View

The Deployment, Logical, Process, and Implementation Views are tied together by implementing the basic stimulus/response use case. A client makes an HTTP

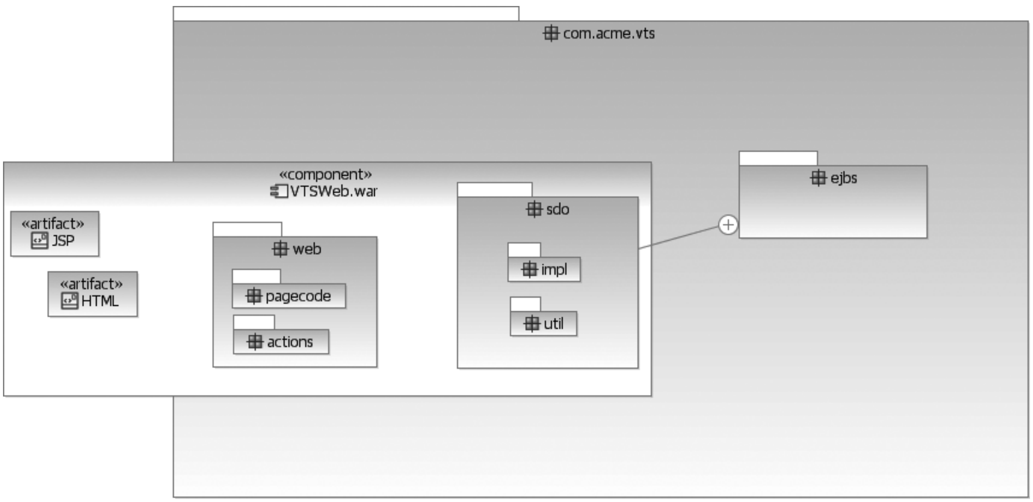

Figure 12–7 The Implementation Model Overview

request to a server for a Web page. The server examines the request and determines which application or resource needs to be loaded and executed. Some resources result in business logic processing, while others simply display static data. When the processing is complete, the application is responsible for composing a response, usually in the form of an HTML Web page that can be rendered in place of the previous Web page. This response page contains new information and options for the user to invoke or request. By assembling an entire collection of these Web pages, each specialized to display and accept information that is part of the application, an entire business process can be implemented.

In this chapter, we focus on one architecturally representative use case: `Manage Time`. This use case by far is the most frequently invoked and the one most viewed by all the actors of the system. As a result, it is critical to implement this use case effectively and to ensure that it meets all of the overall design goals, including the ease-of-use feature.

Many different templates can be used for use case specifications. The format used in this chapter shows more robust steps in the flows. This avoids the stimulus/response style flow that Web applications can create. Also, this style of use case specification is good for situations when there are complex or robust alternate flows. The style or format that you ultimately use is up to you and the level of formality to which your organization adheres.

The following example gives a summary of the written specification for the `Manage Time` use case.

Use case name: `Manage Time`

Actor: `Employee`

Goal: The employee wishes to submit a new request for vacation time.

Preconditions: The employee is authenticated by the portal framework and identified as an employee of the company with privileges to manage his or her own vacation time.

Main flow:

1. The employee begins by selecting a link from the intranet portal to the VTS.
2. The VTS uses the employee's credentials to look up the current status of all the employee's vacation time requests and outstanding balances. Information is displayed for the previous 6 months and up to 18 months in the future.
3. The employee wants to create a new request. The employee selects one of the categories of vacation time with a positive balance to use.
4. The VTS prompts the employee for the date(s) and time for which to request vacation time. The employee should have access to a visual calendar to help select and compare chosen dates.

5. The employee selects the desired dates and hours per date (e.g., four hours might indicate a half-day vacation time request). The employee enters a short title and description (no more than a paragraph in length) so that the manager will have more information with which to approve this request. When all the information is entered, the employee submits the request.
6. If the submitted information is incomplete or incorrect or does not pass validation, the Web page is redisplayed, with the errors highlighted and documented.
7. The employee has an opportunity to change the information or cancel the request.
8. If the information is complete and passes validation, the employee is returned to the main VTS home page. If the employee's vacation time requests require manager approval, an e-mail is immediately sent to the manager(s) authorized to approve the employee's requests.
9. The vacation time request is placed in a state of pending approval.
10. The manager responds to the e-mail by clicking on a link embedded in the e-mail or by explicitly logging into the intranet portal and navigating to the main VTS home page.
11. The manager may be required to supply necessary authentication credentials to gain access to the portal and VTS application.
12. The VTS home page lists the manager's own vacation time requests and outstanding balances but also has a separate section listing requests pending approval by subordinate employees. The manager selects each of these one at a time to individually approve or deny.
13. The VTS displays the details of the requested time and prompts the manager to approve or disapprove the request. If the request is disapproved, the manager is required to enter an explanation. Once the manager submits the result, the internal state of the request is changed to approved or rejected.
14. Whether a request is approved or rejected, an e-mail notification is immediately sent to the employee who made the request. The manager's screen returns to the main VTS home page, and the manager may approve other outstanding requests, make a request for him- or herself, or simply leave the VTS application.

Alternate flow: `Withdraw Request`

Goal: The employee wants to withdraw an outstanding request for vacation time.

Preconditions: An employee has made a vacation time request, and that request has yet to be approved or denied by an authorized manager. See also main flow preconditions.

1. The employee navigates to the VTS home page through the intranet portal application, which identifies and authenticates the employee with the privileges necessary for using the VTS.

2. The VTS home page contains a summary of vacation time requests, outstanding balances per category of time, and the current status of all active vacation time requests for the previous 6 months and up to 18 months in the future.
3. The employee selects a vacation time request to withdraw, one that is currently pending approval.
4. The VTS prompts the employee to confirm the request to withdraw the previously submitted vacation time request.
5. The employee confirms the desire to withdraw, and the request is removed from the manager's list of pending approvals.
6. The system sends a notification e-mail to the manager.
7. The system updates the request state to withdrawn.

Alternate flow: `Cancel Approved Request`

Goal: The employee wants to cancel an approved vacation time request.

Preconditions: The employee has a vacation time request that has been approved and is scheduled for some time in the future or the recent past (previous 5 business days). See also main flow preconditions.

1. The employee navigates to the VTS home page through the intranet portal application, which identifies and authenticates the employee with the privileges necessary for using the VTS.
2. The VTS home page contains a summary of vacation time requests, outstanding balance per category of time, and the current status of all active vacation time requests for the previous 6 months and up to 18 months in the future.
3. The employee selects a vacation time request to cancel, one that is in the future (or recent past) and has been approved.
4. If the request is in the future, the employee is prompted to confirm the cancellation. If the request is in the recent past, the employee is prompted to confirm the cancellation and provide a short explanation. If the employee approves the cancellation and provides the required information, an e-mail notification is sent to the manager, and the state of the request is changed to canceled. The time allowances used to make the request are returned to the employee. The employee can also abort the cancellation, effecting no changes to the current requests.
5. The employee is returned to the main VTS home page. The summaries are updated to reflect any changes made to the employee's outstanding vacation time requests.

Alternate flow: `Edit Pending Request`

Goal: The employee wants to edit the description or title of a pending request.

Preconditions: An employee has made a vacation time request, and that request has yet to be approved or denied by an authorized manager. See also main flow preconditions.

1. The employee navigates to the VTS home page through the intranet portal application, which identifies and authenticates the employee with the privileges necessary for using the VTS.
2. The VTS home page contains a summary of vacation time requests, outstanding balances per category of time, and the current status of all active vacation time requests for the previous 6 months and up to 18 months in the future.
3. The employee selects a request to edit, one that is pending approval.
4. The VTS displays an editable view of the request. The employee is allowed to change the title, comments, or dates. The employee can also choose to delete or withdraw this request.
5. The employee changes request information and submits the changes to the system.
6. If the employee withdraws the request, the VTS prompts for confirmation before withdrawing the request. If changes are made only to the information, the changes are accepted, and the screen returns to the main VTS home page. If there are errors or problems with the information changes, the VTS redisplays the editing page and highlights and explains all problems.

This use case description contains quite a bit of information about the proposed VTS and how it is expected to be used by a typical employee. It is for the most part a functional description of the system from the viewpoint of the `Employee` actor. Indeed, this is exactly what a use case is supposed to be. Unfortunately, this description is insufficient to even begin analysis with. What is needed are the nonfunctional requirements and much more information about the domain. Nonfunctional requirements typically include requirements on the environment, performance, scalability, security, and so on. Domain knowledge can be in the form of discrete requirements, for example:

- All employees work eight-hour days.
- Each employee's vacation time requests are subject to the restrictions of each employee's primary work location in addition to overall company policies and restrictions.
- Vacation time request validation rules are defined and owned by the HR department.

These types of requirements or knowledge may be found embedded in use case specifications, or they can be captured as discrete line item requirements. This type of domain knowledge may also be referenced by commonly accessible documents. For example, the detailed policies and rules for validating a vacation time request are part of a company's employee manual and most likely available through a variety of sources (e.g., intranet, forms, documents, new employee orientation presentations, and so on). A project's requirements set would simply reference these existing documents rather than try to duplicate them.

12.3 Construction

The most important task when analyzing a potential Web application, as with most types of software applications, is the identification of the system's entities and processes. The entities and processes of an application represent concepts in the business domain and are ideally independent of an architecture, but not necessarily so. In a Web application, one critical set of artifacts is the navigation map and Web page definitions. Roughly speaking, these elements correspond to the classic analysis stereotypes of entity, controller, and boundary. We'll begin with a Web-centric model: the User Experience (UX) model.

The User Experience Model

The UX model [1, 2] is one example of capturing the user interface elements of a Web application at a sufficient level of abstraction so as to express a concrete navigational map between the Web pages in the system, while ignoring the styles (fonts, sizes, colors, and so on) and other user interface specifics that are best developed later in the process. Figure 12–8 shows a high-level fragment of a UX model that describes the screens used to implement our primary use case. In this model and diagram there are two key stereotypes, `«Screen»` and `«Form»`. A `«Screen»` stereotyped class represents a complete unit of user interface displayed to an end user or, roughly speaking, a Web page. Some screens contain HTML form elements, which are used to submit user-entered information back to the server. These stereotyped classes are always contained by a screen and when submitted result in navigation to another screen in the system. Directed association relationships indicate navigational pathways through the screens.

While Figure 12–8 is most useful for understanding the navigational flow through the screens of the system, it provides little in the way of screen content. More detailed diagrams that contain screen content are also useful at this level of abstraction. Figure 12–9 shows the content of the `VTSHome` screen. Attributes of a `«Screen»` stereotyped class indicate discrete data values, typically strings or simple types easily rendered in HTML. The `employee name` and `current date` attributes are probably used in a header or at the top of the screen. The `message` attribute is used to display an informative or error message after an

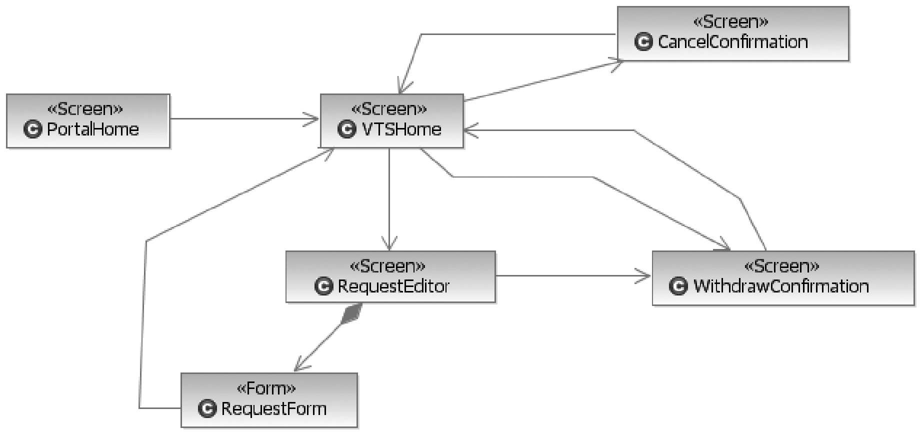

Figure 12–8 A High-Level User Experience Model

action has taken place. For example, when the employee completes a new vacation time request and returns to the home screen, the message might say something like "Your request requires the approval of your manager, who has been notified by e-mail."

Often content in a screen is multivalued and complex. To accurately represent this type of content, we define another class stereotype, `«Content»`, that when applied to a class identifies a coherent bundle of information. Content items are often used as line items in a list. In Figure 12–9, the classes `Request` and `PendingApproval` are modeled as content items contained by the home screen. The screen potentially displays multiple instances of each. Content

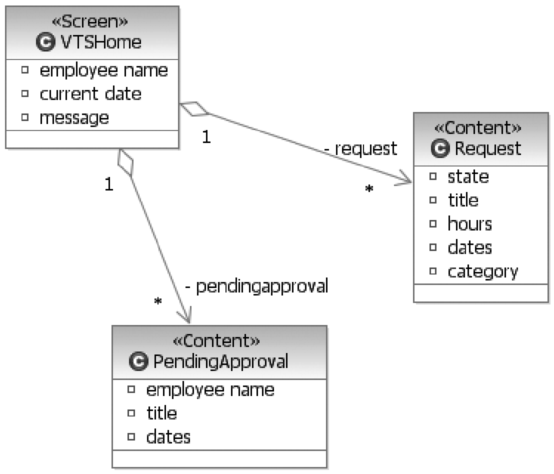

Figure 12–9 A Detailed View of the `VTSHome` Class

classes define attributes that are like the attributes of a screen, things that can be easily rendered in HTML. It is not important at this high level of the model to worry about actual data types. What is important is to simply define them with a name and short description.

It is interesting to note that even at this early point in user experience design, some decisions are being made that will have a significant impact on the final design. By specifying the content of the `VTSHome` page not only to contain the summary of current vacation requests for the employee but also to have this same page used by the manager to view pending approvals, we are implying that this screen is the home page for both employees and managers. Nowhere in the use case specifications nor the nonfunctional requirements was it stated that employees and managers should use the same home screen. Such an assumption is not without reason since managers are also employees and can do all the things (with respect to the VTS) that employees can. Decisions like this are exactly what the UX model is intended to document. Ultimately, the final design will determine the implementation; however, from the logical (and user experience) point of view, employees and managers share the same home screen.

The Analysis and Design Models

In analysis, we want to capture the entities and processes of the system. Since we know that this system will be implemented as a Web application, we are less concerned with boundaries because they are explored in the UX model. The analysis model is a first attempt at identifying the elements that will make up the solution space but using the vocabulary of the domain. This means that most of the elements of the analysis model will have names that correspond to things and activities that are described in the requirements and are recognizable by the domain and end users.

One easy way to get started is to do a simple noun-verb analysis of the use case specifications and related requirements documents. Important nouns tend to represent classes in the model, while action phrases (verbs) tend to be represented by operations on those classes. Attributes and relationships represent natural intrinsic properties of the classes captured in the model. Figure 12–10 represents an initial start, focusing on the main domain classes and concepts represented in our primary use case.

There are several important points to consider in this class diagram. First is the idea that vacation time requests are separate and distinct from vacation time grants. A grant in this context represents available time for the employee to draw on when requesting time off. Grants are administered by the HR department and determined by company policy. In the context of this use case, however, their pri-

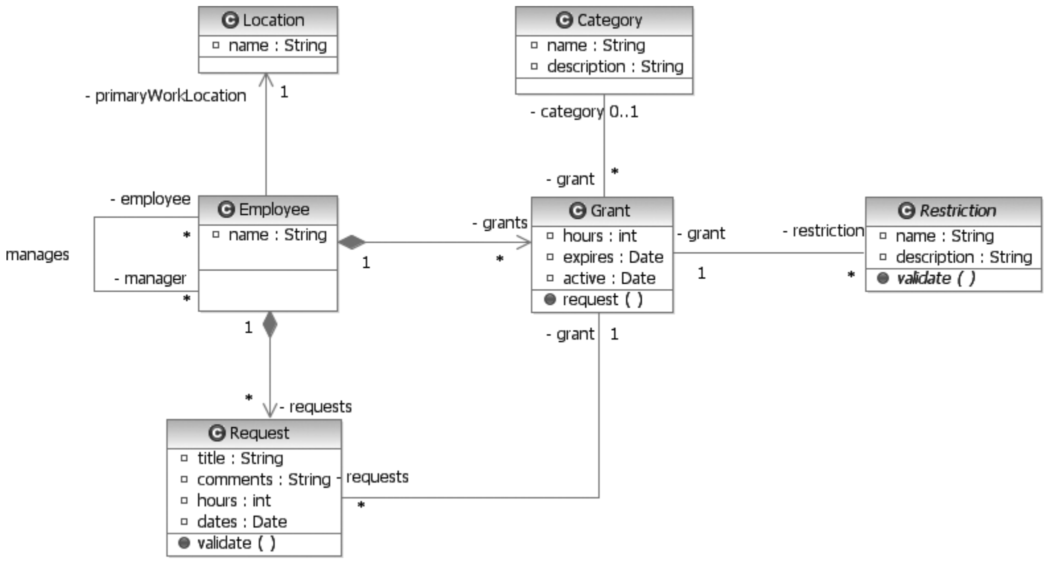

Figure 12–10 An Analysis Model Class Diagram Supporting the Primary Use Case

mary responsibility is to create new vacation time requests. Their persistent properties are the number of hours that can be requested per calendar year and when the grant expires (if at all).

Another important decision captured in this diagram is the idea that a manager *is an* employee. While it may sound obvious to everyone familiar with the popular usage of these common business terms, such assumptions cannot always be made when developing software. The class diagram goes further to say that an employee can have multiple managers. This is not meant to be a statement on the company hierarchy and organization, but rather in this context it means that an approval of an employee's request for vacation time may come from multiple sources. For example, in some companies, high-ranking executives have dedicated personal assistants. A personal assistant may be delegated to make vacation time decisions for the executive.

Other inherent properties of an employee are the name and primary work location. Note that at this level of abstraction and required usage, an employee name need only be managed with a single string data type. Other applications that manage employee information will most likely require separate values for the title and the first, last, and middle names, but in this application those distinctions are not required since the only place the employee name is used is in the display of the home screen. As a general rule, unless the problem absolutely requires a complex solution, don't propose one.

Another interesting and important aspect of most analysis models is the lack of identifier definitions. In nearly every implementation of a system with persistent entities, the concept of a unique ID is present. For example, most companies assign an employee ID to each employee. This ID uniquely defines that particular employee and is often reused if the employee leaves and then returns to the company. The need for the ID is clear given that there is no guarantee of the uniqueness of names. Yet in our model, there are no ID attributes defined. That is because in most analysis models, it can be safely assumed that IDs and other properties necessary to implement the analysis with the architecture will be added as required. Unless there is a very specific business need for information like this, there is no need to specify it during analysis. For example, suppose we had in our proposed system another actor called `Auditor`, who was responsible for browsing all the vacation time requests and checking for regulatory compliance; that actor would in fact need and expect employee IDs to cross-reference with other employee data. In this case, it would be important to explicitly specify this attribute of an employee.

Some interesting elements in the model can be elaborated with a state machine. For example, the `Request` class has a defined state machine, as shown in Figure 12–11. As far as state machines go, it is not that interesting; however, it is significant. It tells us that state is important for a `Request` and that it is well defined. In the diagram shown in Figure 12–11, we see three transitional states and four final states defined. The paths through these states are very simple and in this diagram are unlabeled.

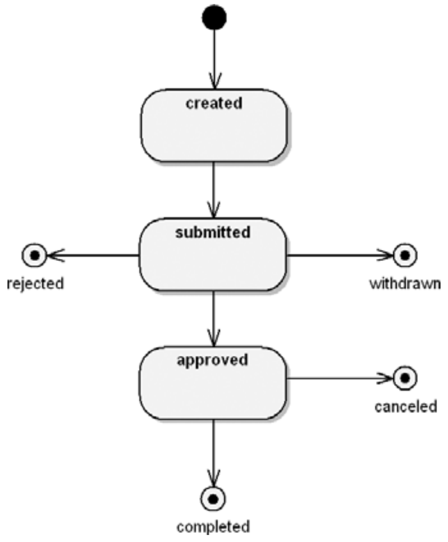

Figure 12–11 The State Machine for the `Request` Class

By far the most difficult part of the analysis was creating the elements related to ensuring that any requested vacation time passed all the rules and restrictions of the company and primary work location. Not much is stated in the use case specifications about these rules; instead, the requirements simply reference internal company documents as current examples. The requirements do state that there is to be a rules-based system managed by the HR department. This statement alone indicates that the approach must be flexible and manageable by users whose skill sets are not in computers or algorithms. This is important because one obvious and very flexible mechanism for implementing a general-purpose rules-based system is to enable the capture and interpretation of a flexible scripting language like JavaScript as a rule. Most algorithms, complex or simple, could be implemented with JavaScript and easily interact with persistent entities and objects. The principal problem with this is, of course, that the typical HR clerk, being charged with the responsibility of maintaining these aspects of the system, does not typically have JavaScript writing as a primary skill.

Long before we choose an implementation solution or even a strategy, we need to first identify the higher-level abstract elements that belong in the model. For now we can create a single class called `Restriction` and place on it all the responsibility for validating a vacation time request against the company and location-specific policies. A deeper understanding of the rules and regulations for vacation time is needed. Unfortunately, not much about the inherent structure of a rule is documented in the use case specifications, and only specific instances of the rules are captured in the company documents. Therefore, analysis leads us to study in detail and look for patterns in the current set of rules. Analysis activities like this often require interaction with domain experts.

After a careful examination of the rules implemented in company policy, and after speaking with the domain experts in the HR department, we can assemble a summary of the major rule types.

- An employee can't take more than *X* consecutive days of leave for *Y* type of grant.
- Vacation time of type *X* cannot be taken when directly adjacent to a company or location-specific holiday.
- Vacation time of type *X* is limited to *Y* hours per week or month.
- Vacation time may not be granted when there are only *X* number of employees scheduled to work from the list *Y* of employees.
- Vacation time may not be granted on these dates: *X*.
- Vacation time of this type is limited to certain days of the week: {M, T, W, Th, F, Sat, Sun}.

This more detailed understanding of potential rules for vacation time request approval will lead to some important changes to our model. It is clear that

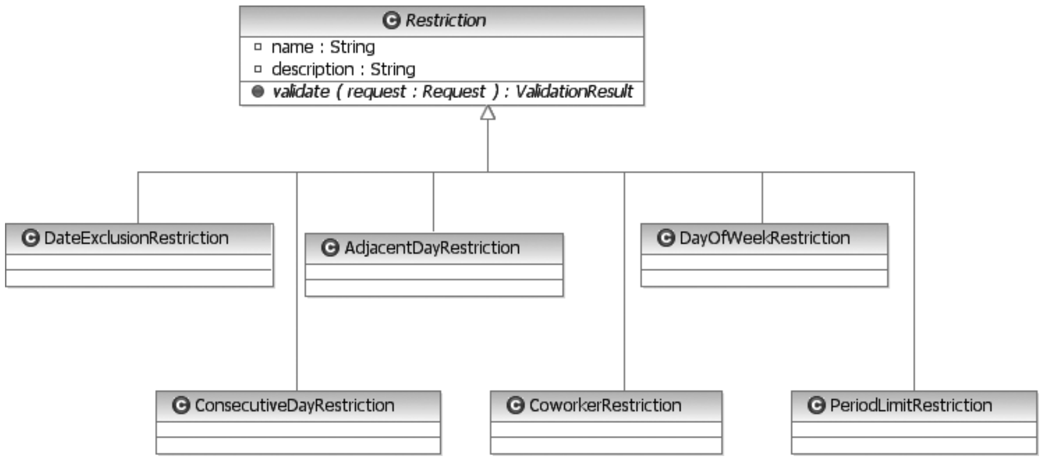

Figure 12–12 The `Restriction` Class Hierarchy

implementing restrictions is varied and requires specific, individually set information. In our attempt to formulate a solution strategy, we will create specializations of the now abstract `Restriction` class, one for each type of rule in the system, as shown in Figure 12–12.

Each specialization is required to implement the `validate()`, method which accepts a `Request` object as a parameter. From the `Request` object a `validate` algorithm can navigate to most of the information it needs (`Employee`, `Grant`, `Location`) to validate the request. Given the varied nature of restriction types, not all of the existing information is sufficient to make a validation. Therefore, each specialization will manage a set of properties or relationships to objects that cannot be derived or navigated to via the `Grant` object. For example, the `CoworkerRestriction` class will need to manage the list of coworkers as well as identify the minimum number of employees allowed to be scheduled for work (Figure 12–13). Rules like this are often associated with safety issues (e.g., there must always be at least one employee on duty who is trained in first aid).

The abstract `validate()` method initially returned just a simple Boolean indicating a pass or fail evaluation of the request. But if we look at the requirements a little closer, we see that the user needs to be notified with an explanation if he or she tries to submit an invalid vacation time request. Since the rules for validation

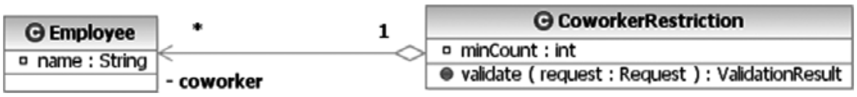

Figure 12–13 The `CoworkerRestriction` Class and Properties

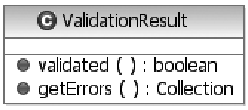

Figure 12–14 The `ValidationResult` Class

are now encapsulated in each of the `Restriction` specializations, it seems appropriate that these classes provide a mechanism to return this information to any calling process. Also, since more than one violation can occur, the results of a validation should be accessed as a collection. The result of these requirements leads us to the creation of a new class, `ValidationResult` (Figure 12–14), which is returned by the `validate()` method.

The `ValidationResult` class is actually quite simple. It defines two methods: `validated()`, which returns a simple Boolean indicating a validated request, and `getErrors()`, a method that returns a collection of string messages, each corresponding to a detected problem with the request. At the analysis level, it is sufficient to keep the class this simple. Let the activities design resolve the details of implementation. The important point here is that it must be possible for a process to validate a vacation time request, make each error result available, and provide a simple means of determining validity.

Another observation of the rules system that is evolving here is that restrictions may not be associated with a particular `Grant` or `Employee` object, but rather, are broadly applied to a `Location` or `Category` of `grants`. Thus our earlier analysis model must change (Figure 12–15). Establishing these relationships makes it easier for the HR department to apply a broad class of common rules without having to duplicate them for each employee. While it is clear this information can be navigated to during validation, the motivation for design change is

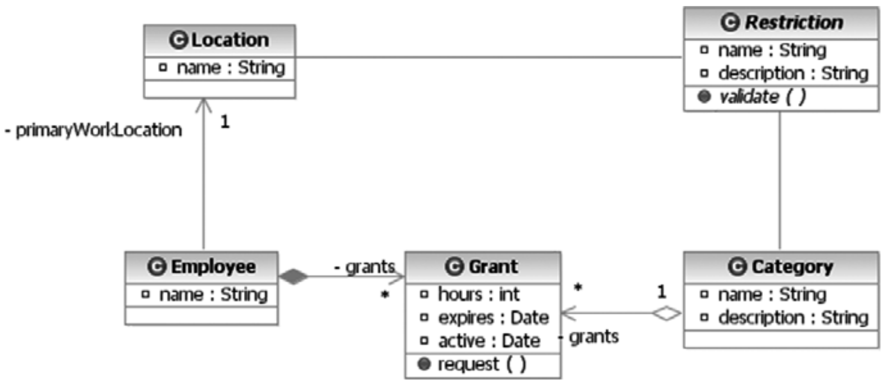

Figure 12–15 Independent and Direct Relationships among `Restriction`, `Location`, `Grant`, and `Category`

based on the overriding desire to make the system easier to use, and in this case the ease of use will be improved significantly for the HR personnel.

Now that the restrictions and rules are better defined, it may be time to revisit some of the HR use cases because now we know more information about the type and nature of information that will need to be managed by the HR department with respect to an employee's vacation time. Although beyond the scope of this simple chapter, a set of Web page screens that prompt for the unique and specific values required by each concrete type of restriction could be envisioned. In such a mechanism, it would be necessary for `Restriction` objects to publish their required parameters (i.e., the *X* and *Y* placeholders in our rules summary) so that the HR clerk could provide the values in a user interface. A helper class, `RestrictionParameterDescriptor`, is defined to encapsulate each of these values. Concrete implementations of `Restriction` are responsible for providing an array of these parameter descriptor objects. The descriptors could be used to prompt the HR clerk when defining new restrictions for an employee, location, or category. Finally, the abstract `Restriction` class should be able to accept and provide individual parameter values. This is accomplished through simple `put` and `get` parameter methods, keyed off of the parameter name. Figure 12–16 illustrates both classes.

At various points during analysis, it is useful to test the ideas being put forth in the model. However, since we are still independent of an actual concrete architecture for which we can write code, we must analyze our model a little more abstractly. We do this by executing thought experiments on the elements of our model, specifically, trying to understand in moderate detail how specific scenarios will play out given the structures and behaviors documented in the model. The area of our model dealing with restrictions is a good example of the need to verify that at least the typical and most important use cases can be implemented this way with realistic data.

A useful pair of tools that the UML provides us is object diagrams and communication diagrams. Our task here is to evaluate the ability of our model to validate a new vacation time request. We want to ensure that the behavior can be accomplished according to the requirements and with the set of defined classes and

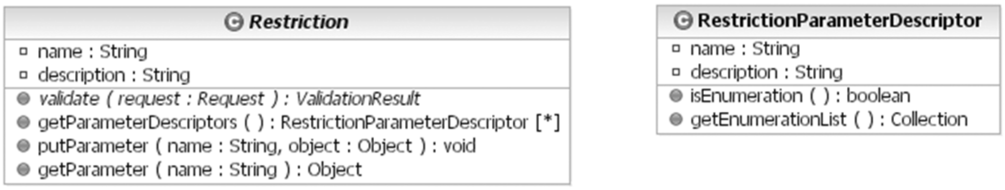
Restriction
name : String
description : String
validate (request : Request) : ValidationResult
getParameterDescriptors ( ) : RestrictionParameterDescriptor [*]
putParameter (name : String, object : Object) : void
getParameter (name : String) : Object
RestrictionParameterDescriptor
name : String
description : String
isEnumeration ( ) : boolean
getEnumerationList ( ) : Collection

Figure 12–16 Making Restrictions Easier to Manage Programmatically and through a User Interface

objects of our model. In this thought experiment, we use these diagrams to understand how a new request for an employee named Jim who works in the North Factory might be executed.

Figure 12–17 shows the object diagram of a set of objects for our hypothetical employee, Jim, who wants to use a vacation time grant given as a bonus for completing his work on time. The figure indicates with dependencies other object instances that can be accessed or navigated to. The instances have real-life names, appropriate for the scenario, and are followed by a colon and the type or class of object they are. This diagram has been annotated with a few freeform rectangles to help emphasize and make clear the three sources of restrictions: `Grant`, `Location`, and `Category`.

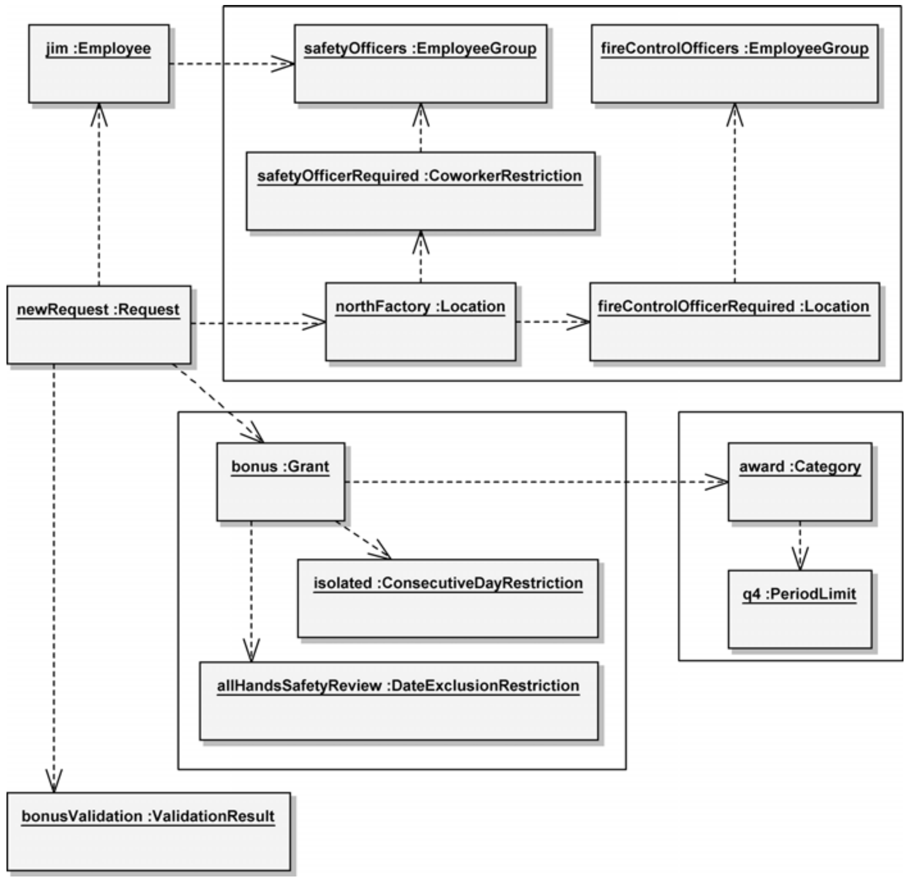
jim :Employee
safetyOfficers :EmployeeGroup
fireControlOfficers :EmployeeGroup
safetyOfficerRequired :CoworkerRestriction
newRequest :Request
northFactory :Location
fireControlOfficerRequired :Location
bonus :Grant
award :Category
isolated :ConsecutiveDayRestriction
q4 :PeriodLimit
allHandsSafetyReview :DateExclusionRestriction
bonusValidation :ValidationResult

Figure 12–17 An Object Diagram Describing a Request Validation Collaboration

In Figure 12–18, we use a communication diagram to examine the communication paths for a Request validation. The lines represent communication paths between object instances, not structural relationships like those found in class diagrams. We examine the behavior of this collaboration by looking at the numbered messages that travel over the communication paths. The main coordinator of the behavior is the Request instance, which does most of the work in this scenario. The general flow is to get and validate the various restrictions for the Location, Category, and so forth by calling the validate() operation on each of the restrictions. If a restriction fails validation, its message is appended to the validation result instance.

One interesting flow to follow is that of the CoworkerRestriction. In order for a CoworkerRestriction to validate, it needs to know not only which EmployeeGroup it represents but also which Employee to check compliance for. A quick look at the validate() signature reveals that the Request object is passed in as parameter, and from it we can navigate to the Employee. However, these restrictions also need access to the vacation plans of all the other

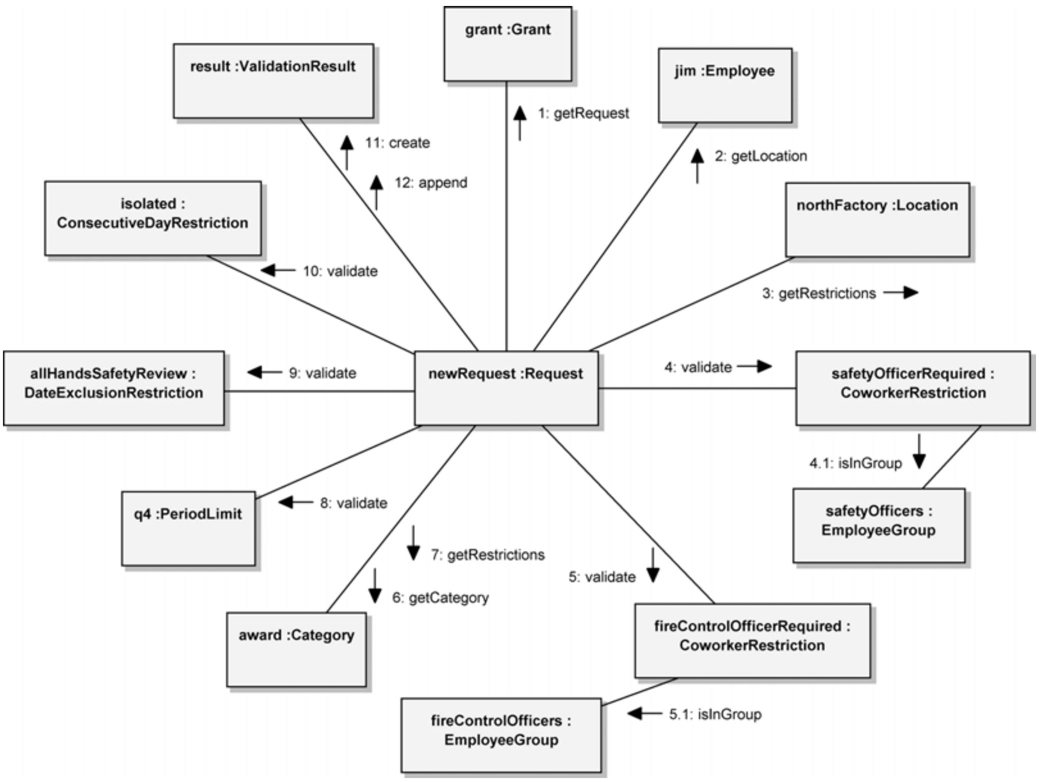

Figure 12–18 A Communication Diagram Describing a Request Validation Collaboration

employees in the group to ensure that at least one employee of the group is scheduled for the location at the request time. So far, our model has not adequately expressed how this can be accomplished. Therefore, as a result of this thought experiment, we need to revisit the model and make sure that these types of restrictions can get the information they need to validate a request.

As discussed in Chapter 6, the beginning and ending of analysis is not a milestone event. Analysis activities often take place throughout the entire development lifecycle, but for the most part they are concentrated at that time when most of the requirements are at least understood. When the analysis model has sufficiently matured, it is time to begin design. This is clearly a vague determination, but that is the reality for most project efforts. The decision to start design activities is often made on criteria that are very local, for example, the history of the project team and organization or the maturity of the architecture.

Now we convert the analysis into a design that can eventually be executed as software. Some design models are identical to executable models (or implementation models), while others are sufficiently abstract as to still require a certain amount of skill and effort during implementation. Web applications in general have a mix of these. Most business tier elements (entities and controllers) map quite closely to the resulting code in the system. Most presentation models, however, require significant effort during implementation before they are ready for the final system.

In this chapter's discussion of the VTS's design, we don't have the space to cover even a brief overview of all the interesting aspects and issues of design in Web-centric applications. Instead, we will discuss just a few of the most important design points, in particular those that are most unique to Web application architectures.

In addition to the basic architectural components of the Web, this project's architect has decided to use the following components and technologies:

- *Client tier*: Any HTML 3.2 capable browser
- *Presentation tier*: Java Server Pages (JSP), Servlets, and Java Server Faces (JSF)
- *Business tier*: Enterprise Java Beans (EJB) 2.x (specifically, Container Managed Persistence [CMP] beans for entities, and session beans for controllers), and Service Data Objects (SDO)
- *Security*: Central Authentication Service (CAS)[2]

The decision was made to use Java-based products in the overall architecture. This is not because Java-based solutions match up with the stated requirements

2. For more information, see www.ja-sig.org/products/cas/.

better than any other existing technology, but rather because they happen to match up better with the organization's current skill sets and tooling, in addition to being well suited to the task. This organization has built several Java-based enterprise applications with the core J2EE technologies successfully in the past. These technologies are familiar, and the tooling required to build and test these types of applications already exists on the developer's workstations. In our situation, there is no compelling reason why the architect should switch entire technology frameworks. There are only two reasons why an architect should even consider such a significant change.

1. Recent experiences demonstrated serious problems with the technology or an inability to meet the needs of the development team, maintenance team, and end users.
2. The technology itself is no longer being supported or is evolving so significantly as to make it appear like a new technology.

In the client and presentation tiers, the primary design goals are to implement an intuitive user interface that has a quick response and is easy to navigate. The interface should not have any dependencies on specific browser versions or features, or to put it more accurately, this application should behave properly on all standard HTML 3.2–capable browsers. Even though the current version of HTML at the time of this writing is version 4.01, the decision to support version 3.2 (and later) is driven not so much by the availability or popularity of browsers but by minimum required functionality. It has been determined that all of the features necessary to implement this application can be accomplished with the older version of the HTML specification. Therefore, since there is no real need to require new browsers and exclude the older versions, the application can support a broader selection of client configurations.

In addition to the core presentation technologies, JSP and Servlets, the architect has decided to use JSF components to assist in the development of the user interface. JSF is an API and custom tag library that contains a number of components for displaying forms in HTML interfaces, accepting and validating input, and assisting in page navigation. An interesting point to note here is a decision not to use the Apache Struts framework. Struts is an older model-view-controller (MVC) framework that is still very popular in JSP applications. Struts and JSF have been shown to work very well together, even if some of their functionality overlaps. While Struts does provide some of the functionality of JSF, its main strong point is its controller features. Struts is especially useful when a Web application implements long (i.e., many Web pages) business processes. It has been determined that our application's business processes are relatively short, and the value that the Struts framework would bring is not significant enough to warrant the extra layer of complexity required to implement it.

The business tier, where most if not all of the application's logic is executed, will run inside a J2EE Web and EJB container. The decision of whether or not the Web and EJB container will run on the same machine, whether they are clustered, or other such configuration information, is not a consideration of the design, except for the fact that such decisions can be made. One advantage of choosing a J2EE-based architecture is the ability to make such deployment choices late, as the system is initially deployed, or to effect these changes as a result of changing demand. The important thing to remember during the design and implementation activities is to not create or code anything that would prevent the normal ability of the J2EE container to shift or deploy components. Such development practices are often implemented in coding or design guidelines that the development team must follow.

The decision was made to use the CMP mechanism for storing all persistent data. There are many reasons to use this mechanism and many reasons why someone might not want to. Other options for persistence include the use of Plain Old Java Objects (POJOs) and a relational object persistence layer like Hibernate, Castor, or the Apache ObJectRelationalBridge (OJB). These options tend to be lighter in weight and more efficient, especially when the application is running on a single node. The decision was made to use the built-in CMP mechanism of the existing container for persistence since the performance needs of this application are not significant enough to warrant another technology or extra layer. Additionally, the use of SDO for managing the flow of data between the presentation and business tiers was adopted to make it easier to manage entity data.

One piece of this application's architecture that is not easily discarded is the use of the CAS for implementing single-sign-on. The idea of single-sign-on is that, while in the same browser, an end user need provide authentication credentials only once to access a whole range of related or unrelated Web applications. It was determined early while creating the vision that the VTS was to be an extension of the existing intranet portal system for the company, and that system currently uses CAS for authentication management. Therefore, the new VTS application will also have to use CAS to identify and authenticate end users.

Entities

The most important part of designing and implementing entities is identifying them. In any given analysis model, any objects with attributes, and even some without any defined attributes, may be designed as entities. The trick is to choose only those classes in the model that really need to be designed as persistent entities. For example, some analysis classes, even those with defined attributes, may merely be transient classes or value classes. For example, the `ValidationResult` class is used only as a return value to the `validate` method. Validations are not logged or recorded in this system, so the `ValidationResult` class does not need to be persistent. When a request is validated, an instance of this class is returned for the caller to examine the results of the validation. When finished, the results are no longer needed or referenced. Therefore, the `ValidationResult` class can remain a simple POJO.

The VTS will use CMP beans to manage all of its persistent entities. CMP has come a long way since it was first introduced. Initially there were a number of problems, specifically with relationships and in performance. However, today's newer specification and improved EJB containers have addressed many of these issues, making CMP a preferred mechanism for managing entities.

Recently, several design patterns have developed around CMP beans. It is generally recommended to use CMP beans to define only local interfaces. A local interface can be accessed only by another bean in the same container. Marshalling data across locally connected beans is much more efficient than doing so over an infrastructure that can support cross-node communication. These objects are instead accessed by a façade object that has local connections to the CMP but also publishes remote interfaces that can be accessed by other session beans and remote clients.

SDO is used to marshal data into and out of the presentation tier. These objects are created and managed by the façade objects in the business tier. SDO is the result of a recent collaboration between BEA and IBM, two big J2EE container vendors, and represents a general agreement to certain best practices when handling data in EJB applications.

Another important design pattern for CMP beans is that they should be designed with minimal, if any, business-level behavior. In fact, it is easiest to consider a CMP definition as little more than a thin wrapping around a database table. All business-level behavior is instead placed in session bean façade objects, which are responsible for orchestrating the state in the various CMP beans to accomplish a unit of business behavior. This has a direct effect on how analysis entities are designed and implemented.

We can begin understanding how all these design-level patterns and conventions get realized in code by looking at the `Category` class. This class is relatively simple because it defines only two persistent properties plus a relationship to instances of the `Grant` class. Fortunately, `Category` does not define any significant behavior at the analysis level. If it did, we would need to remove it from the entity and place it in one of the façade objects. What we are starting with here is a very simple entity.

Design entities are represented by «`Entity Bean`» stereotyped classes. These classes define their persistent attributes; primary key attributes are further stereotyped. The methods of an `«Entity Bean»` class are segregated into local and remote (although in this case `Category` does not define any remote methods, according to our design practices), as well as instance and class level. Each of these corresponds to the actual interface used to define them. In the `Category` class, we need two interfaces, `CategoryLocal` and `CategoryLocalHome`, to define the local methods that can be invoked on instances of the bean and on the home or factory object responsible for creating or finding instances. Figure 12–19 shows a UML-like diagram[3] of the `Category` entity bean with a relationship to the `Grant` entity.[4]

Working with a single model element for each entity makes it easier to understand their relationships; however, in the implementation, any given entity is in fact implemented with a number of separate and distinct classes, interfaces, or configuration files. Our `Category` class, for example, requires two separate interfaces (`CategoryLocal` and `CategoryLocalHome`) and one implementation class (`CategoryBean`), is packaged with other beans in a Java Archive (JAR) file, contributes to the EJB deployment descriptor file, and results in the definition of a table in the database (Figure 12–20). The important thing to realize is that there is typically a large fan-out of model elements as you migrate from analysis to design and implementation. One analysis element will often map to many elements in the implementation, all requiring coordination. Listing 12–1 shows a fragment

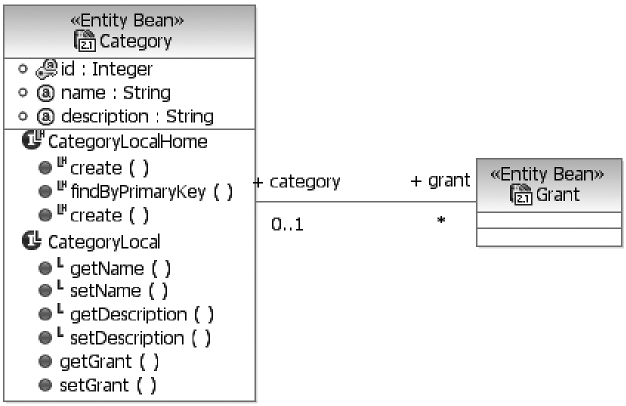

Figure 12–19 A Design Model Representation of the `Category` Class

3. This diagram is a customized view of an entity bean, which in a J2EE system includes the implementation class as well as its interfaces and some configuration information. Also, the use of getters and setters (methods that provide access to object properties) is not an object-oriented convention but rather one dictated by the use of Java and the J2EE framework.

4. Since this is a design model targeting a J2EE architecture, it is not inappropriate to use platform-specific notations.

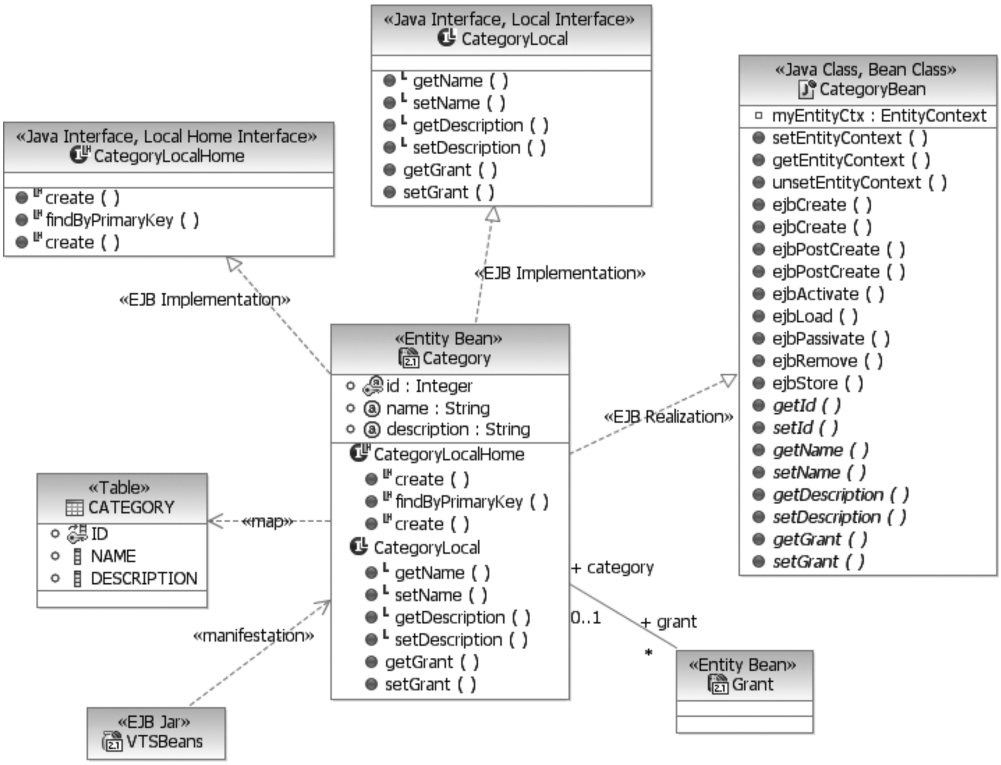

Figure 12–20 Design Model Elements for the `Category` Class

of the EJB deployment descriptor containing elements related to the `Category` entity.

Listing 12–1 A Fragment of the EJB Deployment Descriptor Containing Elements Related to the *`Category`* Entity

```
<ejb-jar id="ejb-jar_ID" version="2.1"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <display-name>VTSEJB</display-name>
  ...
    <entity id="Category">
      <ejb-name>Category</ejb-name>
      <local-home>com.acme.vts.CategoryLocalHome</local-home>
      <local>com.acme.vts.CategoryLocal</local>
      <ejb-class>com.acme.vts.CategoryBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>false</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Category</abstract-schema-name>
      <cmp-field id="CMPAttribute_1107874632507">
        <field-name>id</field-name>
      </cmp-field>
      <cmp-field id="CMPAttribute_1107874633048">
        <field-name>name</field-name>
      </cmp-field>
      <cmp-field id="CMPAttribute_1107874633068">
        <field-name>description</field-name>
      </cmp-field>
      <primkey-field>id</primkey-field>
      <query>
        <description></description>
        <query-method>
          <method-name>findAll</method-name>
          <method-params/>
        </query-method>
        <ejb-ql>select object(o) from Category o</ejb-ql>
      </query>
    </entity>
  ...
  </ejb-jar>
```

Following through with our design guidelines, we define a façade[5] object for the `Category` class. The goal of a session façade is to simplify the interface to do common tasks of a business entity. This often involves the coordination of methods in several objects. The session façade provides a simpler interface to find, create, modify, and delete entities. Fortunately for us, our development IDE provides automation to easily create session façades for entity beans. The resulting façade object is called `CategoryFacade` and, like the entity bean representation, defines functionality at both the remote and local levels and the instance and factory levels. In the model, a stereotyped «`facade`» dependency indicates the semantic connection between the design model–generated façade object and the design model representation of the entity. The façade class provides a number of utility methods for creating, finding, updating, and removing `Category` entities. This essentially acts as a single source manager of instances of this class. Figure 12–21 illustrates such a façade object.

Service Data Objects

Another characteristic of this particular façade implementation is the use of a relatively new and emerging standard, SDO. Simply put, SDO is a mechanism for accessing and manipulating data in a manner that is disconnected from the data

5. See the Session Façade pattern as described in *Core J2EE Patterns* [4].

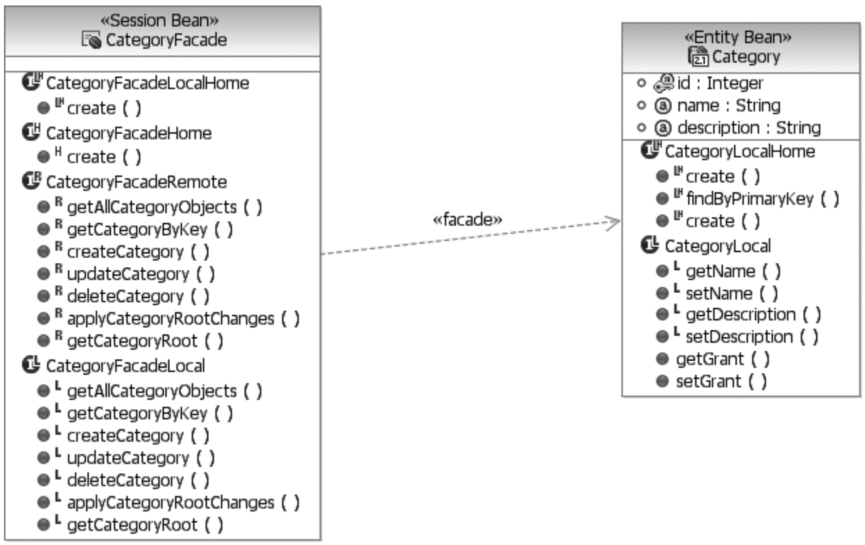

Figure 12–21 A Façade Object Governing Access to Entity Beans

source. This provides a useful means for marshalling data in a system. SDO uses disconnected data graphs and enables clients to retrieve the data graph from a source, manipulate it, and then apply the changes back to the original data source.

The use of SDO makes for a more consistent interface to bean data across the entire application. When the façade classes were generated, the corresponding set of SDO objects was also generated. Each SDO object that is created defines two interfaces: a root object (for the data graph) and an entity interface. The root object represents the conceptual root of the data graph. See Figure 12–22.

The façade object uses the SDO object as the main parameter for its create, retrieve, update, and delete (CRUD) methods. It also provides a method to update any changes made to an entire graph of SDO objects. Typical usage of this façade would be by another session bean acting as a business logic controller, perhaps fulfilling a request on behalf of a Web page. Let's take as a sample scenario the need to update and create vacation time request categories. The responsibility for this particular task falls on the HR clerk role. Such a business logic controller would need to be able to get all the existing categories (usually only a dozen or so)[6] to list on a Web page. It would then allow the user to select one for editing or

6. Arbitrary use of the `getAllObjects()` methods in the façade classes could lead to unexpected and serious performance issues when the system is tested in a real-life situation. Imagine the performance cost of putting into an array the collection of all the customer addresses in any medium-sized business.

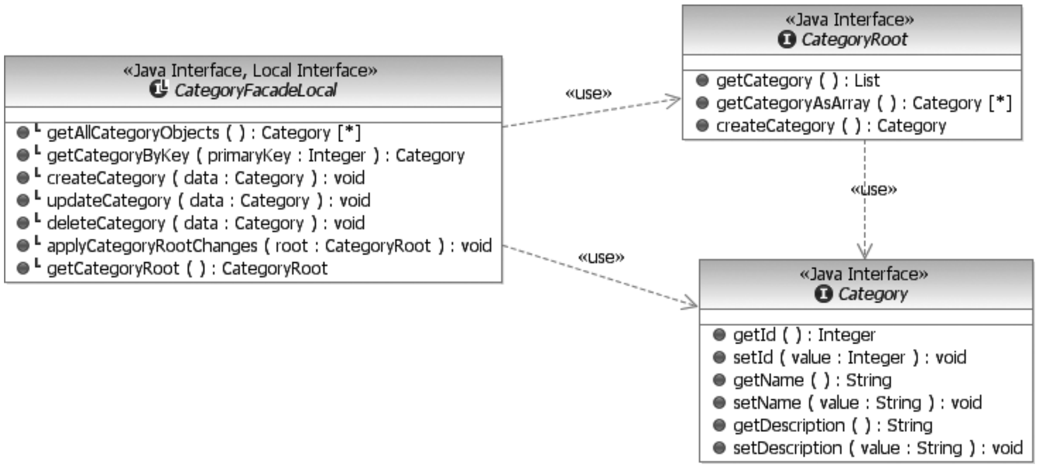
«Java Interface, Local Interface»
CategoryFacadeLocal
getAllCategoryObjects ( ) : Category [*]
getCategoryByKey (primaryKey : Integer) : Category
createCategory (data : Category) : void
updateCategory (data : Category) : void
deleteCategory (data : Category) : void
applyCategoryRootChanges (root : CategoryRoot) : void
getCategoryRoot ( ) : CategoryRoot
«use»
«Java Interface»
CategoryRoot
getCategory ( ) : List
getCategoryAsArray ( ) : Category [*]
createCategory ( ) : Category
«use»
«use»
«Java Interface»
Category
getId ( ) : Integer
setId (value : Integer) : void
getName ( ) : String
setName (value : String) : void
getDescription ( ) : String
setDescription (value : String) : void

Figure 12–22 SDO Objects Used by the `CategoryFacadeLocal` Class

removal. The business logic controller can easily delete a category with a reference to the SDO or its primary key value. Changing the values of a category is similarly handled by changing the values in the SDO and then telling the façade to use its local value to update the persistent entity in the database.

Primary Key Generation

A pervasive issue in all object-relational database systems is the generation of primary keys for entity beans. The availability of natural primary key values like social security numbers or Universal Product Codes (UPCs) are highly dependent on the domain and can't always be assumed to be available. This means that the application must define and implement a strategy for creating primary keys for new entities that have no natural key attributes. In our system, the entities `Grant`, `Restriction`, and `Request` are examples of business objects that have no single attributes that could be safely used as primary key values.

In traditional database systems, a key could be manufactured as a composite of several properties and foreign key values; however, most EJB designs are more efficient with a single primary key value of a primitive type like integer or string. This means that we need to have a strategy for creating primary keys for our entities. Ideally, the same strategy and mechanism could be used for all our entity types that need generated key values.

In this application, we are have selected the Sequence Blocks strategy for primary key generation as described by Marinescu [5]. This strategy refines a very simple approach that essentially creates a new type of CMP entity responsible for managing the next available integer value that can be used as a key for a particular type of entity. The primary problem with this approach is performance, especially in a system where new entities are created often. These performance issues are addressed by managing access to the entity with a session bean that reserves a large block of potential key values. Thus access to the CMP entity holding the keys is slightly reduced, depending on the size of keys reserved by the session beans.

Incorporating this strategy in our application leads to the creation of another entity and session bean with local access by the other entity façade objects (Figure 12–23). The `SequenceEntity` bean has only two attributes: the sequence name (i.e., the name of the entity type requiring a generated primary key) and the next available integer value that can be safely used as a primary key. Because this strategy reserves blocks of key values at a time, it is entirely likely and possible that there will be gaps in the sequence of integer values actually used. Thus, no application using this strategy should make any assumptions about the order or distribution of the primary key values.

The key generator is used by the other façade objects when there is a need to create a completely new instance of an entity. This is accomplished by updating the `createCategory()` method on the façade object to check for an incoming `Category` SDO with a null `id` field. If the `id` field is null, the façade is responsible for generating a fresh key for this category. Failure to do so will result in an exception being thrown.

Getting a new primary key value is as simple as getting access to the `Sequence Session` bean and then calling the `getNextSequenceNumber()` method with the name of the entity as a parameter. The code for the `createCategory` method is shown in Listing 12–2. The `doApplyChanges()` method is an SDO

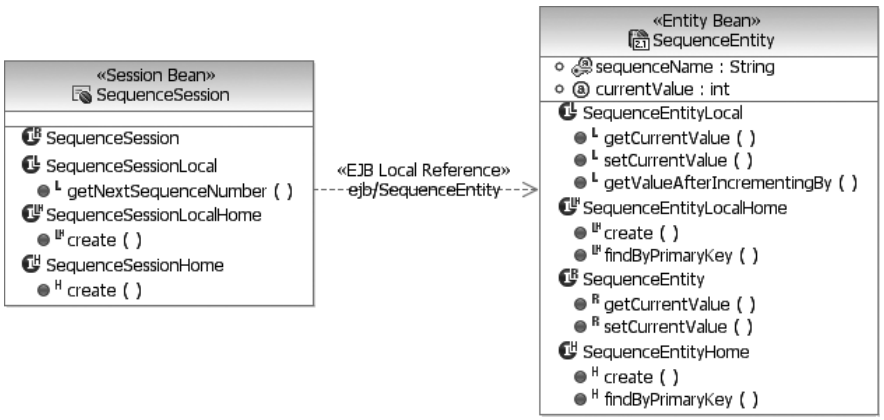

Figure 12–23 Entity and Session Beans for Primary Key Generation

framework implementation method, and generally speaking, the source code is unavailable.

Listing 12–2 Code for the *createCategory* Method

```
public void createCategory(Category data)
            throws CreateException {
    try {
      if( data.getId() == null ) {
        InitialContext ctx = new InitialContext();
        SequenceSessionLocalHome home =
             (SequenceSessionLocalHome)
              ctx.lookup("java:comp/env/ejb/SequenceSession");
        SequenceSessionLocal sequence = home.create();
        int id = sequence.getNextSequenceNumber("Category");
        data.setId( new Integer(id) );
      }
      doApplyChanges(data);
    } catch (Exception ex) {
      throw new CreateException(
          "System error while creating \"Category\".", ex);
    }
  }
```

With this update to all the façade's `create` methods and the use of the `SequenceSession` and `SequenceEntity` beans, new instances of entities can be created without the worry of deriving or computing the required primary key of each instance (Figure 12–24).

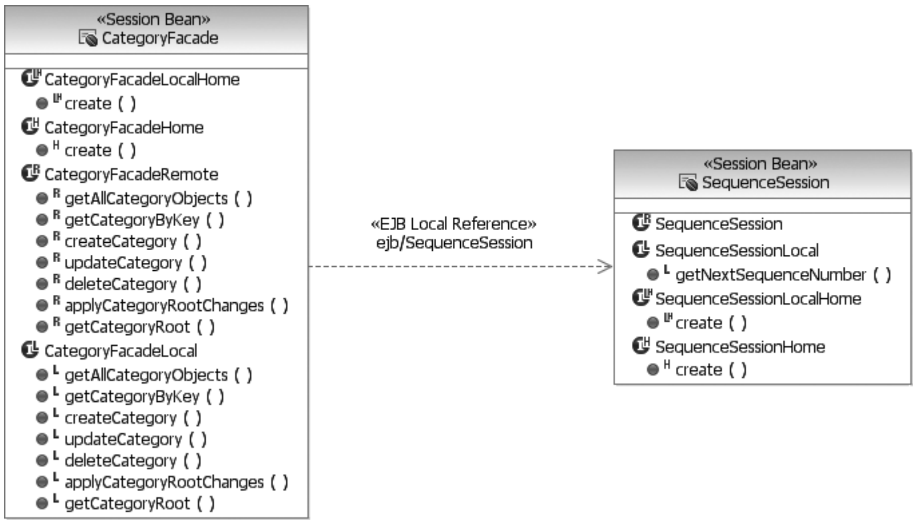

Figure 12–24 The `CategoryFacade` Session Bean Using the `SequenceSession` Bean to Create New Categories

Finders

Another important consideration in EJB entity design is the identification of finder methods. These are methods responsible for searching for specific entity instances that match a given set of criteria. The criteria can be as simple as all instances of vacation time request categories (which typically will result in a list of no more than a dozen instances). The finder may also implement a more complex search, for example, finding all employees whose requests for leaves of absence against a particular category of leave were rejected between March 1 and June 23 of the previous year. Regardless of details, the important point about finders is that the filtering of matching entities occurs on the server, ideally on the database server. Typically, this is the most efficient means of filtering large collections of entities. The only other option is to get all instances of an entity and then, after it has been potentially passed over the wire and reinstantiated in another process, iterate over all the instances and perform the matching tests. This is not the most efficient way to filter large collections.

The idea behind a finder is closely related to that of a SQL `SELECT` statement. EJB finders are expressed in a separate language, the EJB Query Language (EJB QL), which is very similar to SQL but not equivalent. For example, the following EJB QL statement will return all the `Grant` entity instances that are not associated with any restrictions.

```
select object(o) from Grant o where o.restrictions is not
empty
```

Finders can also be parameterized. In the next example, the finder query returns all `DateExclusionRestriction` instances that exclude the supplied date. Parameters are referenced in the query with a question mark followed by the parameter index in the argument list.

```
select object(o) from DateExclusionRestriction o where o.date
is ?1
```

Ultimately, these queries are captured in the main EJB configuration file `ejb-jar.xml` and associated with the Java method declared in the home interface.

Controllers

Even in Web applications there are many approaches to the concept of business logic control. Many Web-centric applications use the model-view-controller (MVC) approach. Currently the most popular framework for implementing strict MVC in J2EE Web applications is the Apache Struts framework. This framework provides a small runtime component along with a set of JSP tag libraries for implementing the user interface and for making it easier to coordinate with the control framework. The MVC design pattern is a widely used and interpreted pattern. There are countless ways to implement the basic pattern even in the context of Web-centric applications. The Struts design and implementation was perhaps the first widely accepted implementation of this pattern for Java-based Web applications.

While the idea of implementing a strict MVC paradigm in the presentation tier sounds reasonable, on closer investigation of the system-level use cases, it is unclear where such control is necessary. In the overwhelming majority of use case scenarios, the functionality required is little more than basic CRUD operations on entities. Orchestrating and coordinating complex business operations, something for which an MVC paradigm is well suited, is just not part of our simple application.

The architect of this system has decided not to use an overt MVC paradigm like that supplied by Struts but instead to rely on the inherent coordination and control supplied as part of the Web pages. For some architects, the decision to not adopt an MVC and/or Struts control framework can be surprising, as many consider MVC critical to all Web-centric applications. We, however, consider the use of middleware, components, or even strategies that are not immediately necessary to the successful release of the application, or not obviously apparent in any existing documentation or future plans for the system, not to be worth the risk.[7]

With this decision made, all of the control constructs will be either in EJB session objects in the business tier, in beans in the server tier, or embedded as tags in the Web pages.

The Web Pages and the User Interface

The design of the Web pages is really separated into two major concerns: (1) the pages themselves, their hyperlinks, and their form fields and (2) their layout. For the first concern, the system architect and information architect collaborate to determine exactly what conceptual pages are required, what should be in them, and how they can be navigated through to accomplish the business goals of the

7. By this same reasoning, some might wonder why the decision was made to adopt EJBs for persistence, when it can be argued that a much simpler and more efficient persistence strategy could also be employed. This decision, like many made in a real-life development project, is often based as equally on technical merit as on organization history and experience.

system. Web pages are typically implemented as JSP pages and contain a mix of presentation data and relationships to business processing beans. The second concern, however, is all about aesthetics and user understanding. The layout of a Web page defines *where* information appears in the page, not *what* information should be there. It focuses on the organization of the page so that what the page presents can be most efficiently understood and worked with by the end users. The remainder of this chapter is devoted to the first concern, as the second one is more about art and visual creativity than traditional analysis and design.

Web page design begins with the UX model. The UX model more often than not describes a very good starting candidate for Web page names, content, and links. Generally speaking, UX «screen» classes map to individual JSP pages. Pure HTML Web pages can be part of a Web application when there is no dynamic content inside the page; however, even in these situations JSP pages are preferred since some client-side state management solutions require the dynamic rewriting of all URL hyperlinks, which can be done only with dynamic pages (see the Client State Management sidebar earlier in this chapter).

Depending on the Web tooling being used, the UX model can usually be easily transformed into a site layout or design model (Figure 12–25), where «screen» elements map to JSP pages and associations map to navigation links, either simple hyperlinks or forms where the user supplies additional data.

When designing the presentation tier, there are two main considerations: how to populate the page's dynamic content and how to invoke the business logic processes during page transitions. This application's architecture is J2EE, and the architect has specified the use of JSF to help integrate page construction with these two tasks. JSF helps simplify the construction of dynamic Web pages. It defines a number of custom tags that are embedded with HTML in JSP pages that are used to invoke methods on business objects and extract data to place in HTML elements that are eventually rendered on the client screen. In addition to the JSF tags, the Java Server Pages Standard Template Library (JSTL) also provides a number of useful tags to work with data in JSP pages.

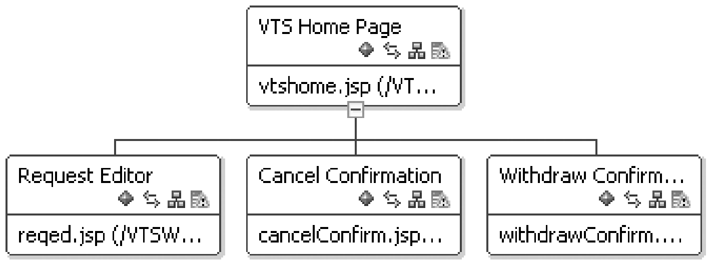

Figure 12–25 The Site Layout Represented in a Design Tool

Populating Dynamic Content

Populating a Web page with dynamic content requires a connection between a Java session bean in the presentation tier and a session bean in the business tier. This should be defined as a remote reference to enable the potential separation of tiers onto different nodes. To populate the VTS home page, we need, in addition to other information, the current employee summary of requests. This summary comes from an `Employee` CMP bean and its related `Request` CMP beans. Since access to CMP beans is governed by generated façade objects, the session bean in the presentation tier needs a remote reference to the façade object. The façade will return SDO objects to the presentation tier session bean, which in turn processes them to produce a simple Java object that can be used directly in the JSP.

In Figure 12–26, the `EmployeeSummarySession` class is a session bean that executes in the presentation tier. Its primary job is to connect to the

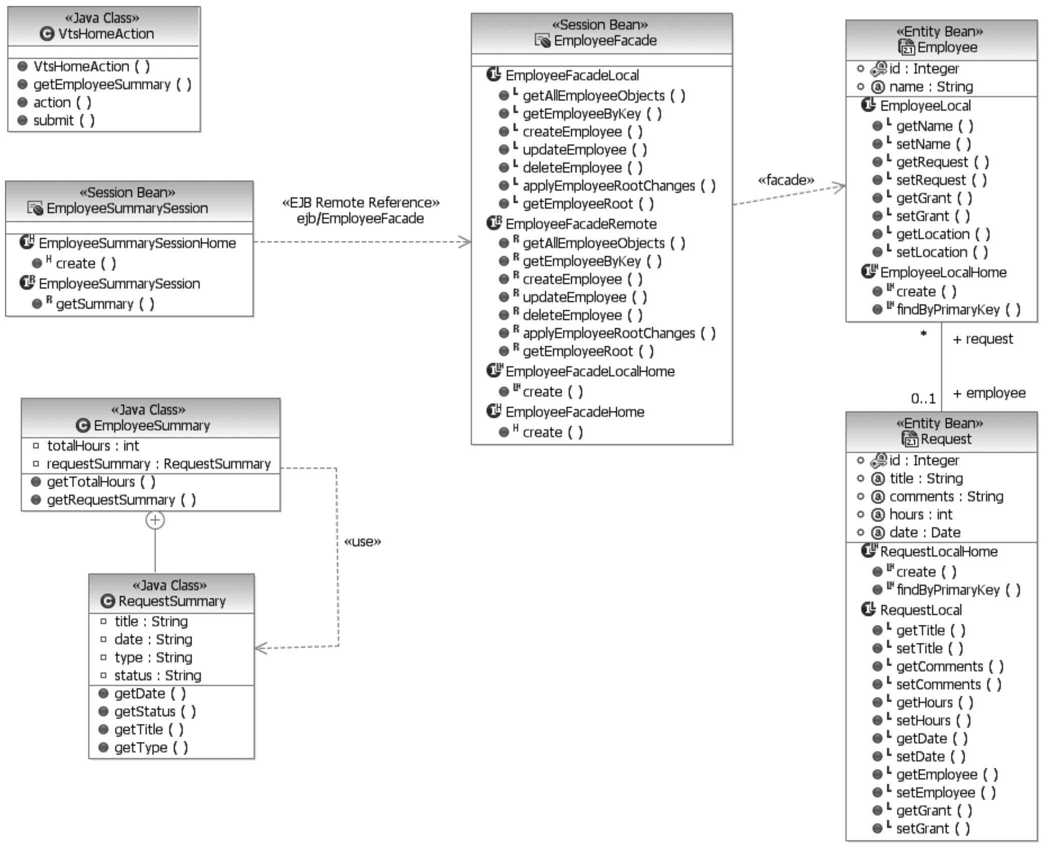

Figure 12–26 Design Elements in the Presentation and Business Tiers

EmployeeFacade and create a summary of an Employee object's requests. This session bean will also collect and make available other employee-specific information that will be used in the VTS home page. The EmployeeSummary class and its public inner class RequestSummary are POJOs used directly in the JSP pages via the JSTL and JSF tags. Part of the JSP source code for the VTS home page is shown in Listing 12–3.

Listing 12–3 JSP Source Code for the VTS Home Page

```
<HTML>
<HEAD>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ page language="java" contentType="text/html;
 charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<jsp:useBean id="vtsHome" class="vtsweb.actions.VtsHomeAction"/>
...
<TITLE>VTS Home Page</TITLE>
</HEAD>
<f:view>
  <BODY>
  ...
  <P>Request Summary</P>

  <c:forEach items="${vtsHome.employeeSummary}">
    <c:url var="delUrl" value="faces/reqed">
       <c:param name="id" value="${leaveRequest.id}" />
       <c:param name="action" value="delete" />
    </c:url>
    <c:url var="edUrl" value="faces/reqed">
      <c:param name="id" value="${leaveRequest.id}" />
      <c:param name="action" value="edit" />
    </c:url>
    <tr>
      <td>${leaveRequest.date}</td>
      <td>${leaveRequest.type}</td>
      <td>${leaveRequest.status}</td>
      <td><a href="${delUrl}">delete</a></td>
      <td><a href="${edUrl}">edit</a></td>
    </tr>

  </c:forEach>

  <P><BR>Outstanding Grants</P>
  ...
  </BODY>
</f:view>
</HTML>
```

The VTS home page, like all of the JSP pages in our application, uses a number of custom tags in several different tag libraries. The three sets of tags that are used the most are the core JSF tags, the JSF HTML tags, and the JSTL tags. These are imported and associated with prefix identifiers at the beginning of the JSP source.

A JSP tag is used to bring in a reference to a presentation tier bean that is used for all access to business state and acts as a local controller for actions in the page. The bean is implemented as a POJO. It is referenced in the context of the JSP with the local reference `vtsHome`.

```
<jsp:useBean id="vtsHome" class=
    "vtsweb.actions.VtsHomeAction"/>
```

This bean can be used in JSF and JSTL custom tags to both extract business data and also invoke business methods. In our home page example, the JSTL `forEach` tag iterates over the `employeeSummary` collection of the bean, which contains instances of `RequestSummary` objects, each representing a specific vacation time request.

In general, it is a good practice to insulate JSP code and logic from the actual EJBs with a simple bean object that can be paired up with a specific page. This object is responsible for organizing and making available all the discrete values of data that can be placed in the JSP. This will also make it easier for the creative team members responsible for the final layout of the page, as they will not have to deal with the sometimes confusing technical requirements of managing EJBs.

Invoking Business Logic

Work is accomplished in Web applications primarily during page transitions. Logic is triggered by user requests to view or navigate to the next page. The next page may or may not be the actual page that the user requested. Depending on the outcome of the business logic, the desired page may get switched to a different page based on the current client's state. For example, if a user submits a request for vacation time but forgets to fill in a required field, the system will not return to the main home page as the user might expect but will return to the editing page and most likely display a warning message indicating the need for the required information.

In a JSF application, the navigation paths are defined by a configuration file called `faces-config.xml`. This file, among other things, defines the managed beans that can be referenced in JSP pages (e.g., `VtsHomeAction`). A navigation rule defines the allowed transitions from one page to another, based on an outcome value. This outcome value is computed within one of the managed beans, and when returned, the `faces` framework will use it to determine which page to load and construct as a response. Listing 12–4 shows the navigation rules in `faces-config.xml`.

Listing 12–4 Navigation Rules in *`faces-config.xml`*

```
<navigation-rule>
  <from-view-id>/reqed.jsp</from-view-id>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/reqed.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/vtshome.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

The rule in Listing 12–4 states that when the button or hyperlink component on `reqed.jsp` is activated, the application will navigate from the `reqed.jsp` page to the `vtshome.jsp` page if the outcome referenced by the button or hyperlink component's tag is `success`. Otherwise, the application will return to `reqed.jsp`.

Hyperlinks or form buttons are placed inside the originating page with the use of the JSF HTML tags. In the following command button example, clicking on the button will cause the `submit()` method of the `vtsHomeAction` instance to be invoked because the `action` attribute references the `submit` method of the `VtsHomeAction` backing bean. The `submit()` method performs some processing and returns a logical outcome that is passed to the default navigation handler, which matches the outcome against a set of navigation rules defined in the configuration file.

```
<h:commandButton value="Submit Changes"
action="#{vtsHome.submit}"/>
```

The impact of this design is that all of the presentation tier's logic is expressed in the managed or backing beans that execute in the Web server. These beans accomplish their tasks by creating normal remote EJB references to EJBs in the business tier, which could potentially be running on a different node in the system.

12.4 Transition and Post-Transition

Web-centric architectures, especially Internet-based ones, bring with them their own special implementation and testing concerns, in addition to the usual ones of functionality and general performance. Web applications by their very nature exist in a heterogeneous environment that may be subject to change; the exact client hardware and software configuration can't be assumed. To help address these issues during design, special browser-specific technologies were purposely not used in our VTS application.

Implementing a Web application is, for the most part, the development of server-side software. Nearly all the software written in a typical Web application executes on the server side. Only when custom JavaScript is employed or specialized applets or client-side controls are developed will the developer really need to be concerned with the details of every client hardware and software configuration the application may encounter.

Even when you choose the technologies that are officially supported by the latest versions of the most popular browsers, there is no guarantee that they will perform the same way across browsers and client configurations. Such technologies and related issues include the following.

- *Java scripting*: All browsers do not implement client-side scripting in exactly the same way. Several browsers extend the scripting language with new and proprietary features. Others interpret the specifications slightly differently or have unusual side effects not governed by the official specifications.
- *Style sheets*: Style sheets and other layout-specific functionality depend on the availability of fonts and screen size. Just because you use style sheets doesn't mean that your pages will render in the same way on all client configurations.
- *Frames*: The implementation of frames has been problematic since their introduction. Scrolling, sizing, and targeting issues constantly plague the use of frames in applications. If frames are used in an application, be sure to test them completely, not only with all targeted client configurations but also in scenarios that involve more than just the use of the application being tested.
- *Bandwidth*: As our development tools become more sophisticated and it becomes easier to incorporate elaborate features into our Web pages, there is the chance that some end users will experience extremely slow page transitions due to the volume of HTML and/or the amount of scripting that must execute on the client. If the application under development is to be deployed to an anonymous user base over the Internet, special care must be taken to test and design the pages for low bandwidth and weak client configurations.

This set of issues becomes an ongoing challenge as technologies evolve over time.