Web Application: Vacation Tracking System

Developing a Vacation Tracking System as a web application offers flexibility for employees and managers to manage vacation time effectively. In this fictitious scenario, the company recognizes the need for such a system due to the increasing independence of workers and the challenges managers face in tracking vacation time across multiple projects and reporting lines.

The decision to develop this system as a web application aligns well with the organization's existing infrastructure, likely as an extension of the intranet. This approach provides a natural entry point for users and integrates seamlessly with other systems already in place within the organization.

Key architectural decisions and considerations for this system include:

1. **Web-Based Architecture**: Leveraging web technologies allows for easy access to the system from any location with an internet connection, promoting flexibility and accessibility for users.
2. **User Roles and Permissions**: Implementing role-based access control ensures that employees and managers have appropriate permissions to view and manage vacation-related information.
3. **Data Management**: The system must handle vacation-related data, including employee profiles, accrued leave, requests, approvals, and historical records. Database management and data security are crucial aspects of system architecture.
4. **User Interface Design**: Creating an intuitive and user-friendly interface is essential for user adoption and satisfaction. Design considerations include ease of navigation, clear presentation of information, and responsive layout for different devices.
5. **Integration with Existing Systems**: The system may need to integrate with other enterprise systems, such as HR databases or project management tools, to synchronize employee information and vacation schedules.
6. **Security**: Implementing robust security measures, including encryption, authentication, and authorization, is critical to protect sensitive employee data and ensure compliance with data privacy regulations.
7. **Scalability and Performance**: Designing the system to handle a growing user base and increasing data volume requires considerations for scalability and performance optimization.
8. **Testing and Quality Assurance**: Thorough testing, including unit testing, integration testing, and user acceptance testing, is necessary to ensure the reliability and functionality of the system.

Overall, the development of a Vacation Tracking System as a web application involves addressing various architectural challenges while prioritizing user experience, security, and integration with existing systems.

1 INCEPTION

During the inception phase, the system's requirements are gathered and documented to establish a clear understanding of the project's scope and objectives. This involves creating several key artifacts:

1. **Vision Document**: This document provides an overview of the project's purpose, goals, and stakeholders. It outlines the vision for the system and its expected benefits.
2. **Key Features**: A list of essential features and functionalities that the system must support. This helps prioritize development efforts and ensure alignment with stakeholders' needs.

3. **Use Case Model**: A graphical representation of the system's interactions with users, including actors (such as employees and managers) and use cases (such as requesting vacation time and approving requests).
4. **Key Use Case Specifications**: Detailed descriptions of critical use cases, outlining the steps involved, preconditions, postconditions, and any alternate flows. This provides a clear understanding of how users will interact with the system.
5. **Architecturally Significant Line Item Requirements**: Identification of specific requirements that have a significant impact on the system's architecture. This includes considerations for scalability, security, performance, integration, and compliance with industry standards or regulations.

By compiling these artifacts during the inception phase, the project team can establish a solid foundation for subsequent phases of development. It ensures that all stakeholders are aligned on the project's objectives and sets the direction for designing and implementing the system.

1.1 The Requirements

While the requirement "The system must be easy to use" may seem overly broad and subjective, it reflects a crucial aspect of the project's vision and goals. Here's why:

1. **Alignment with Vision**: The requirement aligns directly with the vision of the project, which emphasizes empowering employees to manage their vacation time effortlessly. Ensuring the system is easy to use is essential for achieving this goal.
2. **User-Centric Focus**: By prioritizing ease of use, the project acknowledges the importance of user experience and aims to create a system that users will find intuitive and accessible. This user-centric approach enhances satisfaction and adoption rates.
3. **End-User Acceptance**: Ultimately, the success of the system depends on user acceptance and adoption. If users find the system difficult to navigate or understand, they are less likely to engage with it effectively. Therefore, prioritizing ease of use is critical for achieving user buy-in and overall project success.
4. **Quality Assurance**: While the requirement may not be easily quantifiable or measurable in traditional terms, it serves as a guiding principle for quality assurance efforts. Development teams can use usability testing, user feedback, and iterative design approaches to ensure that the system meets user expectations for ease of use.

In summary, while the requirement may appear simplistic, it reflects a fundamental aspect of the project's objectives and underscores the importance of prioritizing user experience in system design and development.

let's simplify:

This requirement says the system needs to be easy for people to use. It's a big deal because if the system is hard to use, people won't want to use it. So, even though it sounds basic, it's super important for the success of the project. It means making sure the system is friendly and straightforward for everyone who uses it.

sometimes we include vague features in the requirements not because we can test them like other requirements, but because they help guide decisions about the design. For example, if a system needs users to submit formatted text, we might choose between a simple method using codes or a more complex but user-friendly option like a formatting tool. If ease of use is a top priority, we might lean towards the user-friendly option.

The main goal of this application is to make managing vacation time faster and easier for employees and managers. Before, vacation requests had to go through several manual steps, which could take days. With the new system, it should only take one manual approval from the manager, and some employees might not need manager approval at all.

this system can save time and money for the HR department by automating the process of handling individual time-off requests. Instead of HR manually approving each request, the system will use predefined rules to validate them. HR will still input and update employee vacation data, but they won't need to be involved in the request and validation process anymore.

The system will have these main features:

- It will use a flexible rule-based system to check and approve leave requests.
- Managers can approve requests (optional).
- Users can view requests from the past year and make requests up to 18 months in advance.
- Email notifications will be sent for request approvals and status updates.
- It will utilize existing hardware and middleware.
- Integrated into the existing intranet portal system with single-sign-on authentication.
- Keeps activity logs for all transactions
- HR and system admins can override actions based on rules, with logs for these overrides.
- Managers can allocate personal leave time within set limits.
- Provides a web service for other systems to access employee vacation summaries.
- Interfaces with HR legacy systems to gather employee information and updates.

1.2 The Use Case Model

The use case model includes four actors and eight use cases, shown in Figure 12–1. Each use case represents a significant value to the actor invoking it, such as managing vacation time requests. For example, the "Manage Time" use case allows employees to view, create, and cancel vacation requests, providing significant value to them.

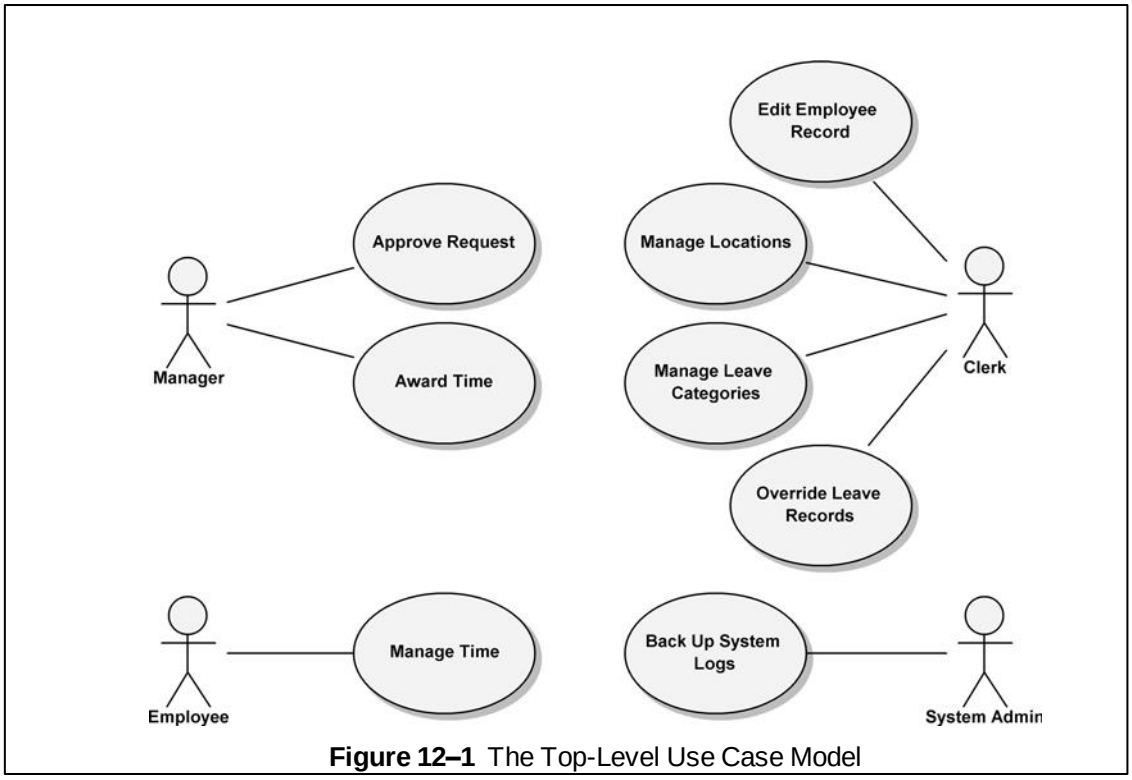

Figure 12–1 The Top-Level Use Case Model

In Web-centric systems, use cases are often described in terms of actor actions and immediate system responses. These scenarios usually correlate closely with specific web pages or screens. In contrast, non-Web applications focus more on the information and interactions exchanged rather than specific user interface elements like web pages.

The system involves four actors:

- **Employee:** The main user of this system. An employee uses this system to manage his or her vacation time.
- **Manager:** Approves subordinates' vacation requests and may award comp time.
- **Clerk:** Manages HR data, including employee records and leave time.
- **System Admin:** Oversees technical resources and logs.

Main use cases include:

- **Manage Time:** Describes how Employees request and view vacation time requests.
- **Approve Request:** Managers respond to vacation requests.
- **Award Time:** Managers grant extra leave (comp time).
- **Edit Employee Record:** HR clerks update employee info, including leave allowances.
- **Manage Locations:** HR clerks handle location records and rules.
- **Manage Leave Categories:** HR clerks manage leave categories and rules.
- **Override Leave Records:** HR clerks override leave request rejections.
- **Back Up System Logs:** System admin backs up system logs.

2 ELABORATION

In software development, the distinction between analysis and implementation phases isn't always clear-cut. Iterative processes are favored because they allow for flexibility, unlike the rigid waterfall model. However, it's crucial to address architecturally significant aspects early on.

In an ideal scenario, system analysis should be independent of the specific implementation architecture. But in reality, prior knowledge of the architecture can influence the analysis. For instance, knowing that a system will be web-based prompts considerations like the need for visual calendars due to browser limitations.

There's no standardized way to quantitatively represent an architecture, but the 4+1 architecture view model is commonly used. Now, let's briefly discuss key aspects of a web application's architecture to provide context for later design decisions.

2.1 The Deployment View

In the deployment view of a web application, we have two main nodes: the server and the client browser. The server has a known network address and listens for HTTP requests on a specific port, usually port 80. When a user requests an HTML resource via their browser, the client browser sends a request to the server. The server may be running various services concurrently, such as other web applications, a database server, or an application server. In Figure 12–2, these nodes, Client and Server, are clearly delineated.

In the Deployment View, the key components include nested nodes within the server node: Tomcat and Cloudscape. Tomcat is a web application execution environment built on the Java platform. Here, the Tomcat node is depicted deploying the artifact VTSWeb.war, which is a Web application archive file. Cloudscape, on the other hand, is an execution environment for a database server capable of executing SQL files. In this view, Cloudscape is shown deploying an artifact named VTS.sql.

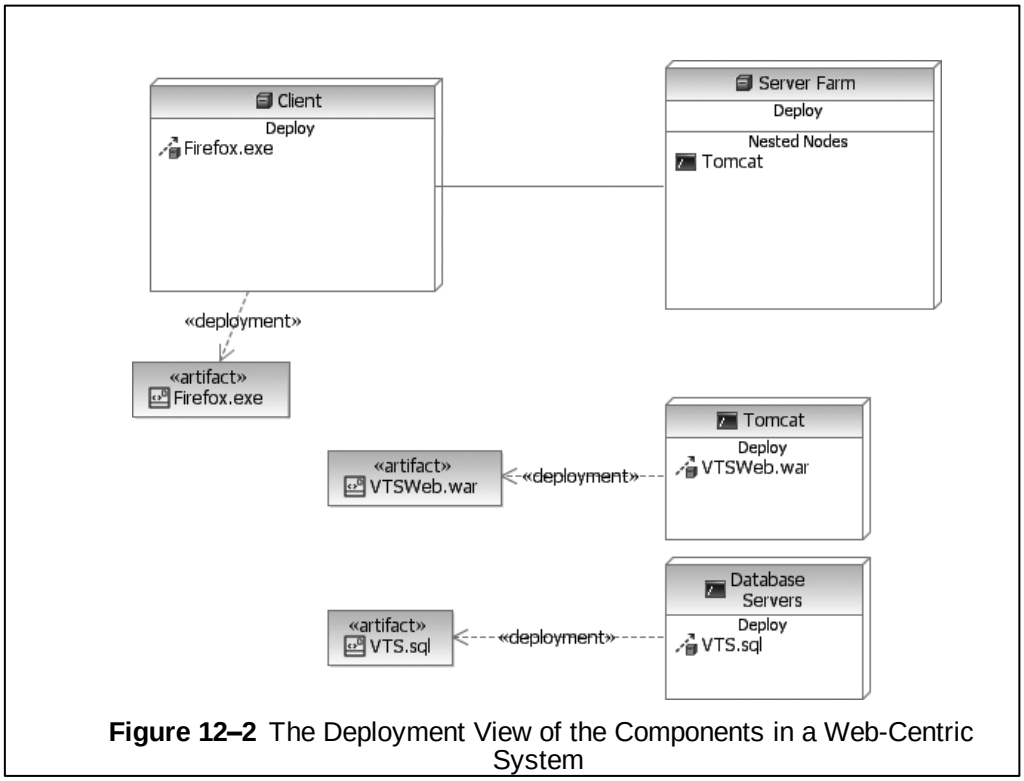

Figure 12–2 The Deployment View of the Components in a Web-Centric System

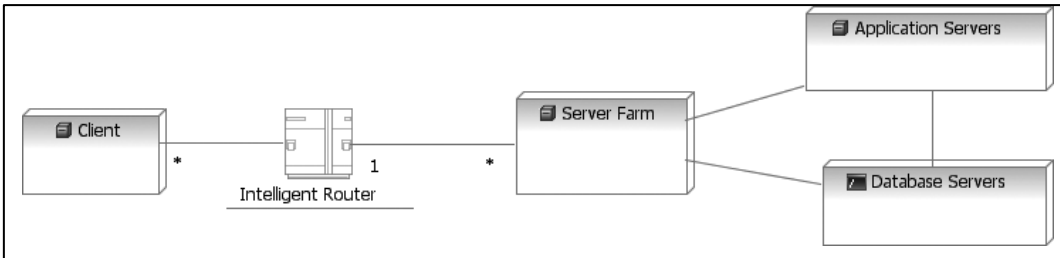

Figure 12–3 A Deployment Diagram for a Web Application Server Farm with Parallel and Redundant Processing Nodes

In Figure 12–2, the Client node is depicted deploying the Firefox.exe artifact, which represents an executable HTML browser application. This client node maintains a communication link with the Server node. This communication link can vary from a dial-up ISP to broadband or wireless connection. The crucial point to focus on is that the client interacts with the nested Web and database servers through this communication link.

Figure 12–3 simplifies a Web system setup. In complex systems, deployment diagrams can get crowded, showing many nodes, servers, and links. To handle this, tools like UML are vital. This abstract diagram shows how a smart router works with a server farm and multiple applications hosted on app servers. Scaling up by adding more processing nodes is a common strategy for web apps.

2.2 The Logical View

In a typical Web app, there are four main parts: the client's browser, a Web or app container, the app's logic, and a database server. The browser (like Firefox.exe) and the app container (like Tomcat) interact. The app (like VTSWeb) and database (like Cloudscape) don't directly communicate with the browser. Instead, they go through the container. This setup boosts security. Web servers, on port 80, handle GET and POST requests in the HTTP protocol. GET requests info, while POST sends user-supplied data or files. HTML-formatted documents usually come back from the server, but it could send data in any format.

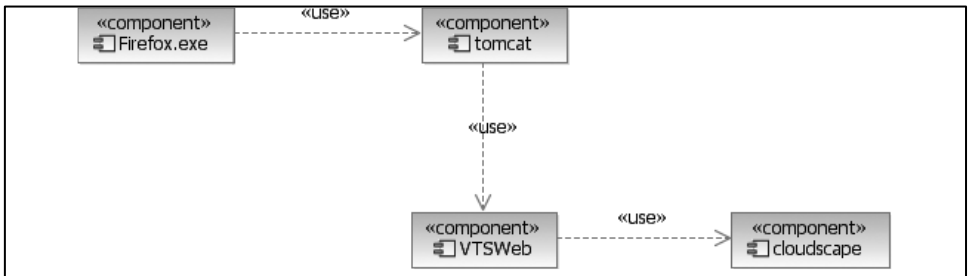

Figure 12–4 The Logical View of the Primary Components of a Web Application

Figure 12–5 illustrates logical classes for both client and server tiers. On the client side, the browser requests Web pages using simple HTTP GET or POST commands. These requests are sent to the Web container, which is always listening on certain ports (like port 80) for incoming HTTP requests. The request includes a resource identifier pointing to a specific page or application and may also have additional parameters in key-value pairs, often containing form data submitted by the user.

When the Web container gets a request for a resource, it figures out which file or application to use. If it's a dynamic resource, the container runs it. Then, it examines the HTTP request info, like parameters and form data, and the application does its job. Usually, this logic happens in a separate application server, like an EJB container, which might access another database server.

In a web app's logic, you have web pages, controllers, and entities. Static web pages sit in a file system, while pages with business logic run in a container. Controllers are often part of components, and databases manage persistent entities.

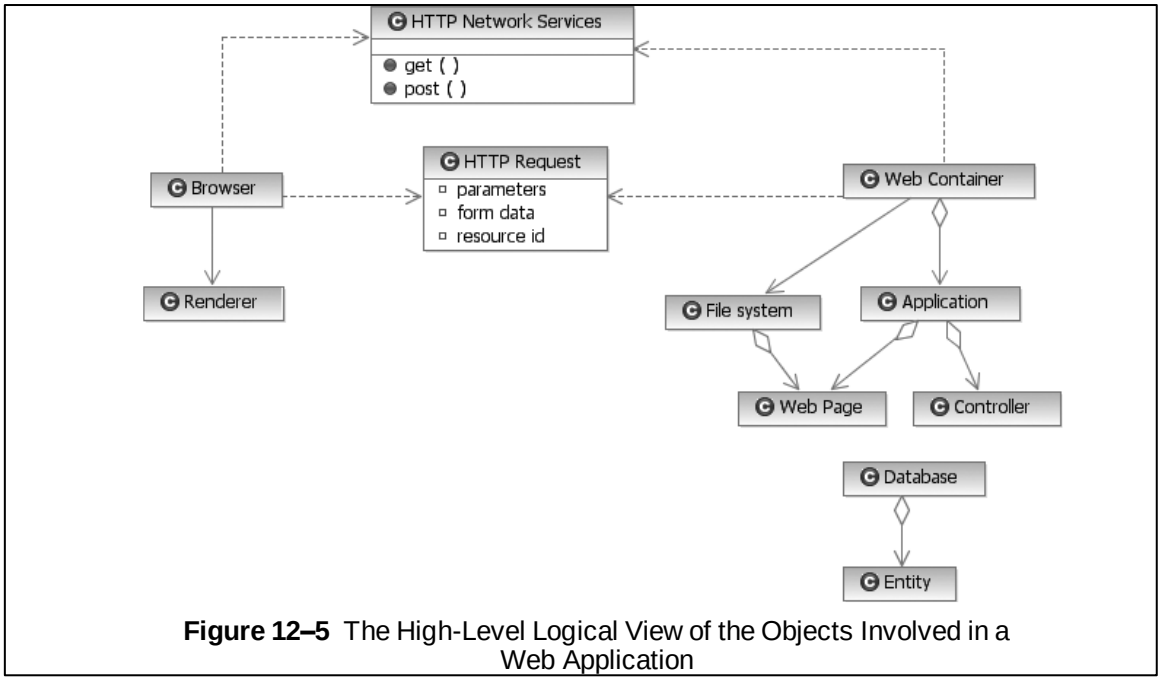

Figure 12–5 The High-Level Logical View of the Objects Involved in a Web Application

2.3 The Process View

In a web app, you typically have at least two processes: one for the client and one for the server. These processes work asynchronously except when handling HTTP requests. There might be other servers like databases or authentication systems involved, either on the same node or distributed across multiple nodes. The layout can be flexible, but clients need web browsers to communicate with the server.

In web app architectures, the client and server operate without a continuous connection. When a client requests a resource (like a web page) from the server, it sends a GET or POST request. The server processes the request, executes any necessary business logic, and sends back the requested resource. After that, the connection is closed. Once the server fulfills the request, it stops working for that client until another request is made. Figure 12–6 shows a client making a simple GET request to the server (Tomcat).

Exactly. Since HTTP is stateless, meaning it doesn't retain information between requests, managing client state in web applications can be challenging. However, web application frameworks offer various tools and mechanisms to handle this issue. For example, sessions and cookies are commonly used to maintain user state across multiple requests, enabling features like shopping carts or multi-step wizards to function smoothly. These tools help the server track and manage the state of individual clients as they interact with the application.

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.

Absolutely, you're spot on. Web application designs need to carefully manage resources and state between page requests due to the stateless nature of HTTP. Opening a transaction in one page and closing it in another can lead to various issues, especially with the short duration between requests. Transactions and locks need to be managed efficiently to avoid performance bottlenecks and potential system failures, especially considering the unpredictable nature of user interactions and network conditions.

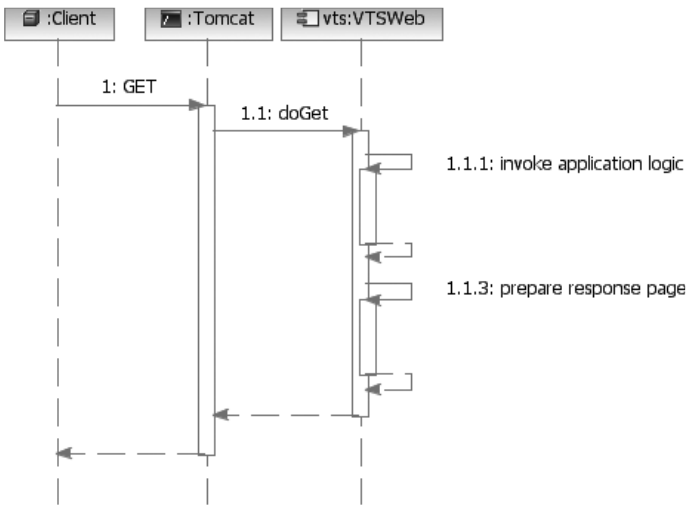

Figure 12–6 A Basic Sequence Diagram of an HTTP `GET` Request

2.4 The Implementation View

In the implementation model for the Vacation Tracking System, the main Web tier component, VTSWeb.war, contains various artifacts like JSP and HTML files, along with Java classes in the web and sdo packages. These components are responsible for processing and invoking the EJB logic in the business tier. The Java components are organized under the com.acme.vts namespace. The Deployment View provides more detailed information about how these components are deployed, while the Logical View elaborates on the nature and responsibilities of each component.

Figure 12–7 shows an overview of the implementation model for the Vacation Tracking System and describes the system's tiers and packages.

2.5 The Use Case View

The Use Case View ties together the Deployment, Logical, Process, and Implementation Views by illustrating the basic stimulus/response scenario. A client initiates an HTTP request to a server for a Web page. The server analyzes the request to determine the appropriate application or resource to load and execute. Some resources trigger business logic processing, while others display static data. Once processing is done, the application generates a response, typically an HTML Web page, presenting new information and options to the user. By stringing together these Web pages, each tailored to handle specific application data and user interactions, the entire business process can be realized.

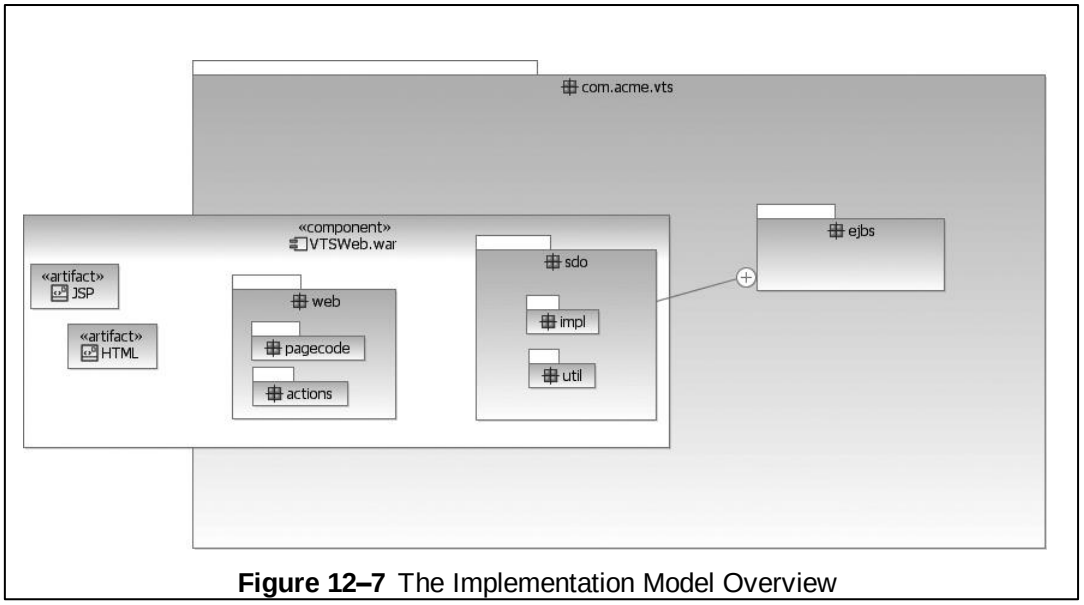

Figure 12–7 The Implementation Model Overview

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

Use Case: Manage Time

Actor: Employee

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

Preconditions: The employee is logged into the company intranet portal and has permissions to manage their vacation time.

Main Flow:

1. Employee navigates from the intranet portal to the Vacation Tracking System (VTS).
2. VTS retrieves and displays the employee's current vacation time requests and balances for the past 6 months and up to 18 months in the future.
3. Employee chooses a category of vacation time with available balance.
4. VTS prompts the employee to select dates and times using a visual calendar.
5. Employee selects desired dates and hours, adds a short title and description, and submits the request.
6. If there are errors, VTS highlights them on the page for correction.
7. Employee can revise or cancel the request.
8. If the information is complete and valid, the employee returns to the VTS home page. If manager approval is needed, an email notification is sent.
9. The request enters a pending approval state.
10. Manager receives email notification and accesses VTS through the portal or email link.
11. Manager provides authentication credentials if required.
12. VTS home page displays manager's own requests and those pending approval by subordinates. Manager reviews and acts on each request.
13. VTS shows request details and prompts manager to approve or deny, with an option to provide reasons for denial.
14. Upon approval or rejection, an email notification is sent to the employee. Manager returns to VTS home page.

This use case ensures employees can efficiently manage their vacation requests while providing managers with a straightforward approval process

Alternate Flow: Withdraw Request

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

Preconditions: An employee has submitted a vacation time request pending approval.

1. The employee accesses the VTS home page via the intranet portal and is authenticated.
2. The VTS home page displays a summary of vacation time requests, including pending approvals.
3. Employee selects the vacation time request they wish to withdraw.
4. VTS prompts employee to confirm the withdrawal.
5. Employee confirms withdrawal, and the request is removed from the manager's pending approvals list.
6. The system sends a notification email to the manager.
7. Request state is updated to withdrawn.

Alternate Flow: Cancel Approved Request

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

Preconditions: Employee has an approved vacation time request.

1. Employee accesses VTS home page and is authenticated.
2. VTS home page displays vacation time requests, including approved ones.
3. Employee selects the approved request they wish to cancel.
4. If in the future, employee confirms cancellation. If in the recent past, employee confirms cancellation and provides explanation.
5. System sends email notification to manager and updates request state to canceled.
6. Employee returns to VTS home page with summaries updated.

Alternate Flow: Edit Pending Request

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

Preconditions: Employee has a pending vacation time request.

1. Employee logs into VTS via intranet portal.
2. VTS home page shows summary of pending requests.
3. Employee selects request to edit.
4. VTS displays editable view of request, allowing changes to title, comments, or dates.
5. Employee submits changes.
6. If withdrawing request, confirmation is prompted. If making changes, changes are accepted, or errors are highlighted for correction.

The use case description provides functional details about how the Vacation Tracking System (VTS) is used by employees, but it lacks essential non-functional requirements and domain-specific information. Non-functional requirements cover aspects like environment, performance, scalability, and security. Domain knowledge includes specific rules and policies relevant to the system, such as employee work hours and location-specific policies.

For instance:

- Employee work hours are typically eight-hour days.
- Vacation time requests must adhere to both company-wide policies and location-specific rules.
- Validation rules for vacation time requests are established and managed by the HR department.

These requirements and domain knowledge may be scattered throughout the use case descriptions or captured separately as discrete requirements. They may also be referenced from existing documents like employee manuals, accessible via intranet or other sources. Instead of duplicating this information, the project's requirements set would simply point to these existing documents for clarity.

3 CONSTRUCTION

The User Experience (UX) model focuses on defining the entities, processes, and navigation map of a Web application. These elements are crucial for understanding the business domain and designing the user interface effectively. In this model, we typically identify:

- **Entities**: Represent the core concepts in the business domain, such as users, products, or orders.
- **Processes**: Capture the actions or operations that users can perform within the application, like creating an account or making a purchase.
- **Navigation Map**: Defines the flow of navigation within the application, detailing how users move between different pages or screens.

These components align with traditional analysis stereotypes like entity, controller, and boundary. Starting with a Web-centric model allows us to focus on designing a user-friendly interface that meets the needs of both the users and the business.

3.1 The User Experience Model

The User Experience (UX) model abstracts the user interface elements of a Web application, focusing on creating a navigational map between different screens without getting into specific design details like colors or fonts. In this model, we have two key stereotypes: «Screen» and «Form».

- A «Screen» represents a complete user interface unit, essentially a Web page.
- Some screens contain HTML form elements, labeled as «Form», used for submitting user-entered data to the server.
- Directed association relationships between screens indicate navigational pathways.

This high-level abstraction helps visualize how users will interact with the application and move between different parts of the system.

Figure 12–8 provides a broad overview of how users navigate through the screens of the system, but it lacks details about the content of each screen. More detailed diagrams, like Figure 12–9, show the specific content of individual screens.

- In Figure 12–9, the attributes of a «Screen» class represent discrete data values, such as employee name, current date, and message.
- These attributes are typically used to populate different parts of the screen, like headers or informative messages.
- For instance, the message attribute might display a confirmation or error message after a user action, like submitting a vacation time request.

These detailed diagrams help designers and developers understand exactly what information needs to be displayed on each screen of the application.

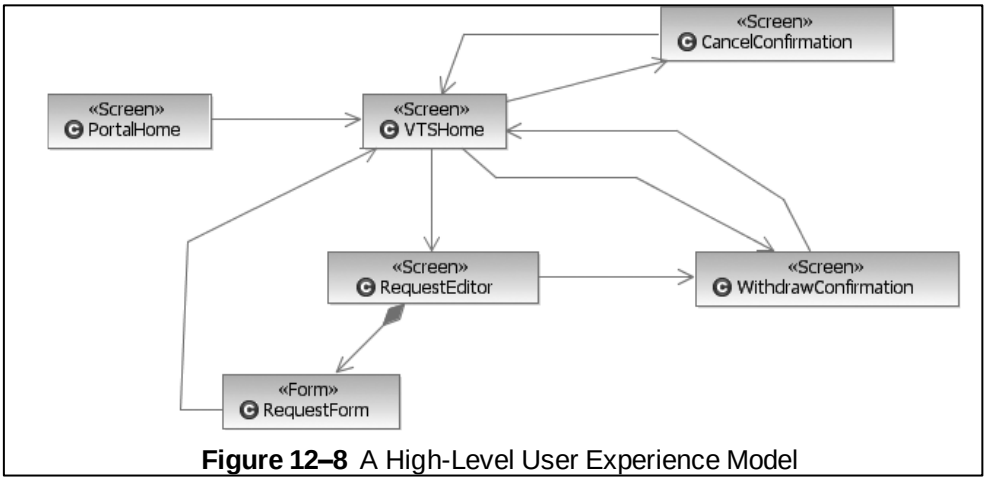

Figure 12–8 A High-Level User Experience Model

In complex screens where content is multivalued, we use the «Content» class stereotype to represent coherent bundles of information, often displayed as line items in a list.

- In Figure 12–9, classes like Request and PendingApproval are modeled as content items contained by the home screen.
- Multiple instances of each content item can be displayed on the screen.
- Content classes define attributes similar to those of a screen, focusing on information easily rendered in HTML.
- At this high level of modeling, actual data types aren't crucial; it's more important to define attributes with a name and short description.

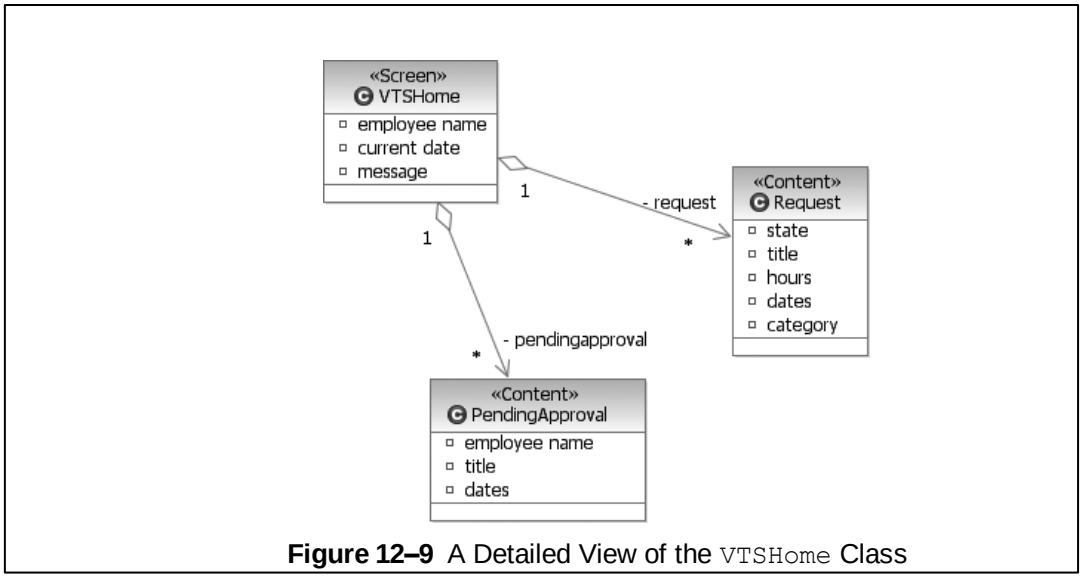

Figure 12–9 A Detailed View of the `VTSHome` Class

3.2 The Analysis and Design Models

In the analysis phase, we aim to capture the entities and processes of the system, focusing less on boundaries due to exploration in the UX model. The analysis model uses domain-specific vocabulary, reflecting elements described in requirements.

- Noun-verb analysis of use case specs and related docs helps identify classes (nouns) and operations (verbs) in the model.
- Attributes and relationships represent intrinsic properties of classes. Figure 12–10 shows initial domain classes and concepts from our primary use case.
- A key distinction is made between vacation time requests and grants. Grants represent available time for employees and are administered by HR, while requests are initiated by employees. Grants persist with properties like yearly requestable hours and expiry dates.

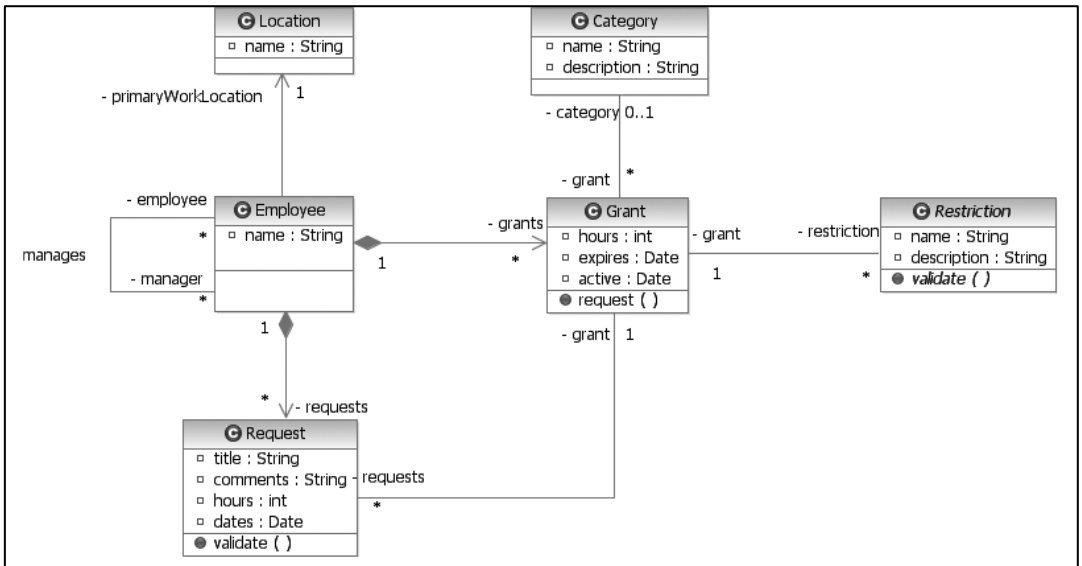

Figure 12–10 An Analysis Model Class Diagram Supporting the Primary Use Case

In this analysis model, a manager is considered an employee, reflecting common business terminology. An employee can have multiple managers, allowing for multiple approval sources. Properties like employee name and primary work location are inherent. Employee names are simplified to a single string, considering the display context. Identifiers like employee IDs are omitted as they're typically added during implementation if needed. State machines, like the one for the Request class, depict well-defined states and transitions, emphasizing the importance of state in managing requests.

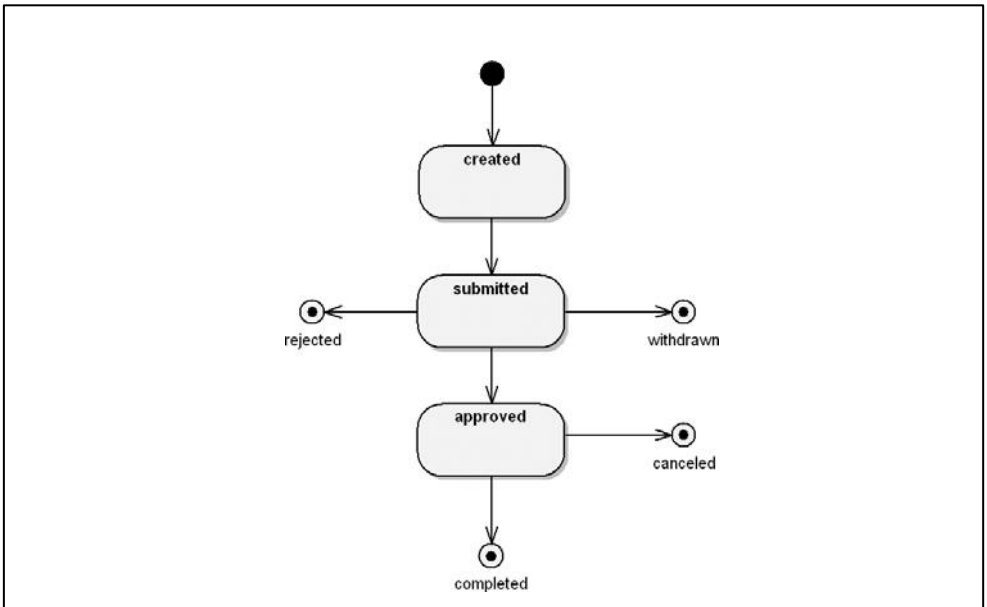

Figure 12–11 The State Machine for the `Request` Class

3.3 Entities

In designing entities, it's crucial to identify classes that truly need to be persistent entities. Not all classes with attributes need to be designed as entities; some may remain transient or value classes. For example, classes like ValidationResult, used only temporarily for method return values, don't need persistence.

The Vacation Tracking System (VTS) will utilize CMP (Container Managed Persistence) beans to manage its persistent entities. CMP has evolved, addressing past issues like relationship management and performance. It's advisable to use CMP beans with local interfaces, optimizing data access within the same container for efficiency.

SDO (Service Data Objects) is employed for marshaling data between tiers, managed by facade objects in the business tier. SDO represents best practices for data handling in EJB applications, developed by industry leaders like BEA and IBM.

An important pattern for CMP beans is to keep them behaviorally minimal, serving mainly as wrappers around database tables. Business logic is centralized in session bean facade objects, orchestrating behavior across CMP beans. This influences how analysis entities are designed and implemented, emphasizing separation of concerns and centralized behavior management.

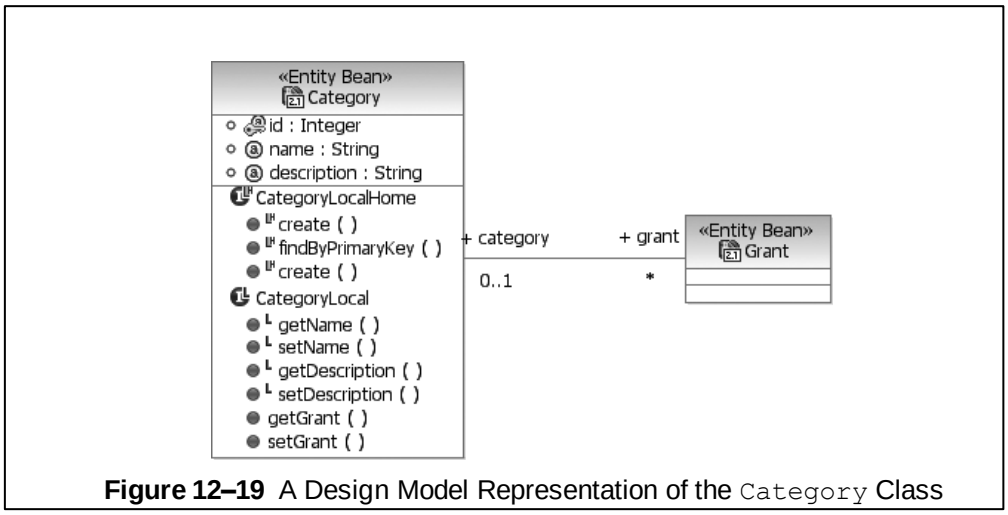

Figure 12–19 A Design Model Representation of the `Category` Class

Let's dissect the Category class to understand how design-level patterns translate into code. Category is a straightforward class with two persistent properties and a relationship to instances of the Grant class. Since Category doesn't define significant behavior at the analysis level, it remains a simple entity.

In code, design entities are represented by classes stereotyped as «Entity Bean». These classes define persistent attributes, with primary key attributes further stereotyped. The methods of an «Entity Bean» class are divided into local and remote interfaces. For Category, we require CategoryLocal and CategoryLocalHome interfaces to define local methods and the home or factory object responsible for creating or finding instances.

The Category entity bean has a relationship with the Grant entity, as shown in a UML-like diagram. While the model may simplify understanding, the actual implementation involves multiple classes, interfaces, or configuration files. For instance, Category requires CategoryLocal and CategoryLocalHome interfaces, along with a CategoryBean implementation class. These are packaged into a Java Archive (JAR) file, contribute to the EJB deployment descriptor, and define a table in the database.

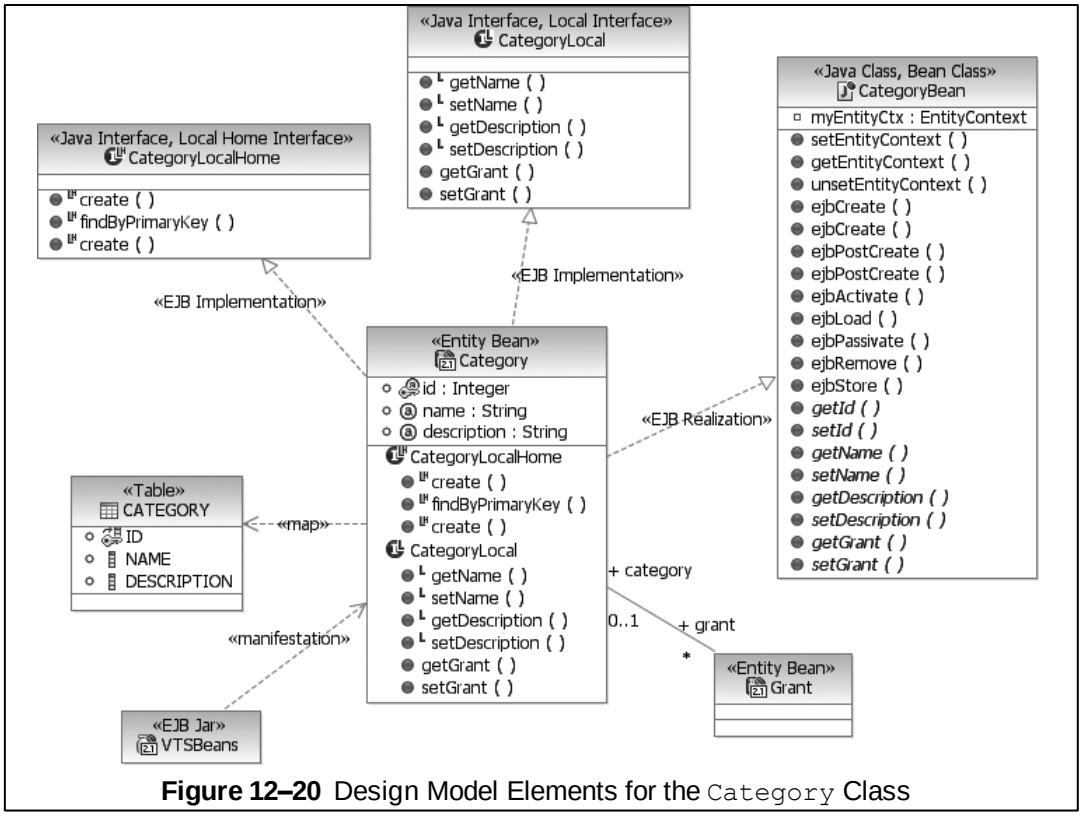

Figure 12–20 Design Model Elements for the `Category` Class

During migration from analysis to design and implementation, a single analysis element often maps to many implementation elements, requiring coordination. For instance, the Category entity in the EJB deployment descriptor involves various related elements.

3.3.1 Service Data Objects

Service Data Objects (SDO) is a relatively new standard used in the façade implementation described here. SDO provides a way to access and manipulate data independently of the data source, which is useful for marshaling data within a system. It operates using disconnected data graphs, allowing clients to retrieve the data graph, modify it, and then apply the changes back to the original data source.

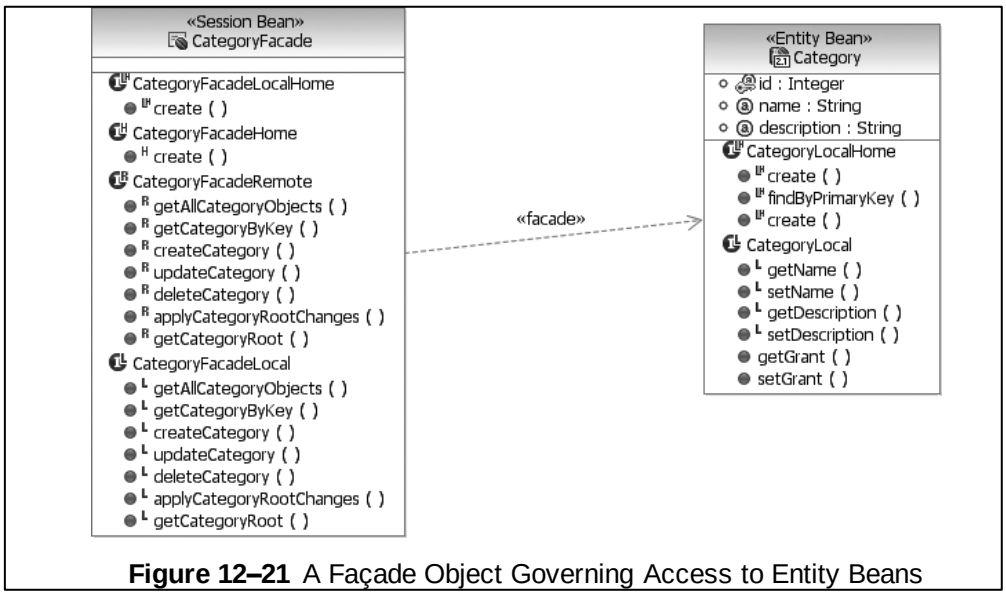

Figure 12–21 A Façade Object Governing Access to Entity Beans

SDO ensures a consistent interface to bean data throughout the application. When the façade classes were generated, a corresponding set of SDO objects was also created. Each SDO object 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. Refer to Figure 12–22 for visualization.

The façade object utilizes the SDO object as the primary parameter for its create, retrieve, update, and delete (CRUD) methods. It also offers a method to update any changes made to an entire graph of SDO objects. Typically, another session bean serving as a business logic controller would utilize this façade, often in response to a request from a web page.

For instance, let's consider the scenario where an HR clerk needs to update or create vacation time request categories. This business logic controller would first retrieve all existing categories (usually a small number) to display on a web page. Then, it would allow the user to select a category for editing or removal. Deleting a category is straightforward, as the controller can simply reference the SDO or its primary key value.

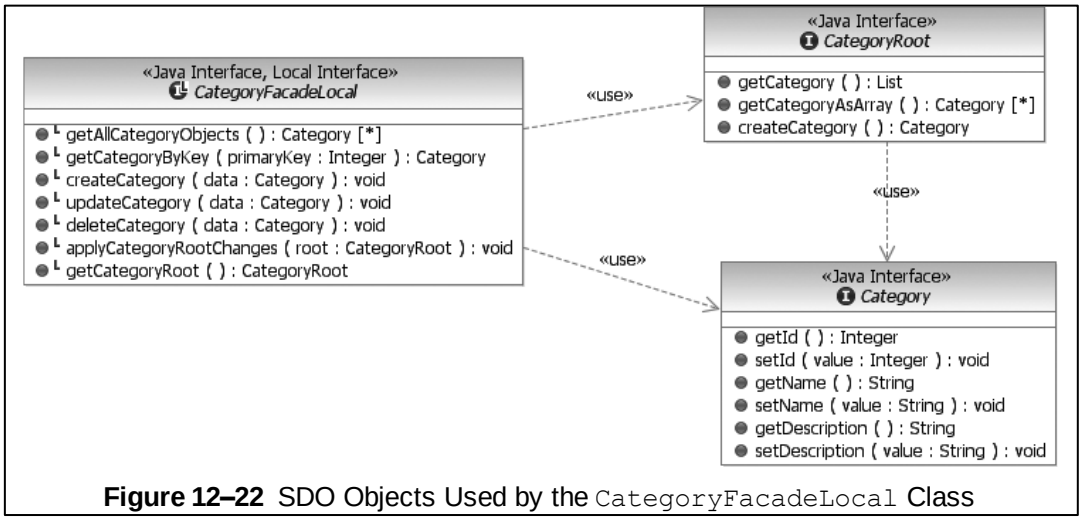

Figure 12–22 SDO Objects Used by the `CategoryFacadeLocal` Class

Similarly, modifying category values involves updating the corresponding values in the SDO and then instructing the façade to use its local value to update the persistent entity in the database. This process ensures that changes made on the web page are accurately reflected in the underlying data store.

3.3.2 Primary Key Generation

Generating primary keys for entity beans is a common challenge in object-relational database systems, especially when natural primary key values are not readily available. In our system, entities like Grant, Restriction, and Request lack single attributes suitable for use as primary keys.

While traditional database systems could construct composite keys from multiple properties and foreign key values, EJB designs typically favor single primary key values of primitive types like integers or strings. Therefore, we need a consistent strategy for generating primary keys across all entities requiring them.

One approach is to use a centralized key generation mechanism, such as a database sequence or an identity column. With this method, the database automatically assigns unique primary key values when new records are inserted, ensuring consistency and avoiding conflicts.

Alternatively, we could implement a custom key generation strategy within the application logic. This could involve generating primary key values based on a combination of factors, such as timestamps, random numbers, or other unique identifiers.

Regardless of the approach chosen, it's crucial to ensure that the selected strategy effectively generates unique primary keys and can scale with the system's growth. Additionally, the key generation mechanism should be transparent to the application code, allowing entities to be created without explicit handling of primary key generation.

The Sequence Blocks strategy for primary key generation, as described by Marinescu, involves creating a separate CMP entity responsible for managing the next available integer value to be used as a key for a specific type of entity. This approach addresses the performance concerns associated with frequent entity creation by managing access to the key-holding entity through a session bean, which reserves a large block of potential key values.

In our application, implementing this strategy involves creating another entity and session bean with local access by other entity facade objects. The SequenceEntity bean, with attributes for the sequence name and the next available integer value, manages the allocation of key values. Since this strategy reserves blocks of key values at a time, there may be gaps in the sequence of integer values used as primary keys. Therefore, no assumptions should be made about the order or distribution of primary key values.

The key generator is utilized by facade objects when creating new instances of entities. For example, the createCategory() method in the facade object is updated to check if the incoming Category SDO has a null id field. If the id field is null, the facade is responsible for generating a new key for the category. Failure to generate a key will result in an exception being thrown, ensuring that new entities are assigned unique primary keys.

“1” The Sequence Blocks strategy involves a CMP entity managing the next available integer value for primary keys. A session bean reserves blocks of key values to improve performance. This approach requires creating a SequenceEntity bean and updating facade methods to generate keys for new entities. Gaps in key sequences may occur, so no assumptions should be made about key order. Failure to generate a key results in an exception.

To obtain a new primary key value, access the SequenceSession bean and call the getNextSequenceNumber() method with the entity name as a parameter. The createCategory method utilizes this approach to generate keys for new entities. The doApplyChanges() method is part of the SDO framework implementation, and its source code is typically unavailable.

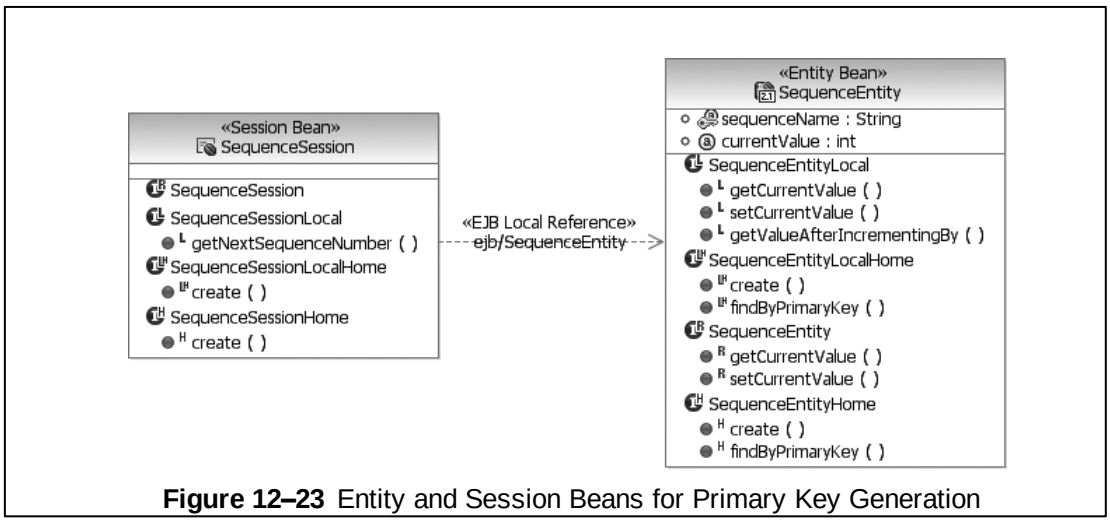

Figure 12–23 Entity and Session Beans for Primary Key Generation

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 these modifications to the create methods in the facade, along with the integration of SequenceSession and SequenceEntity beans, new entity instances can now be created without the need to compute their primary keys manually.

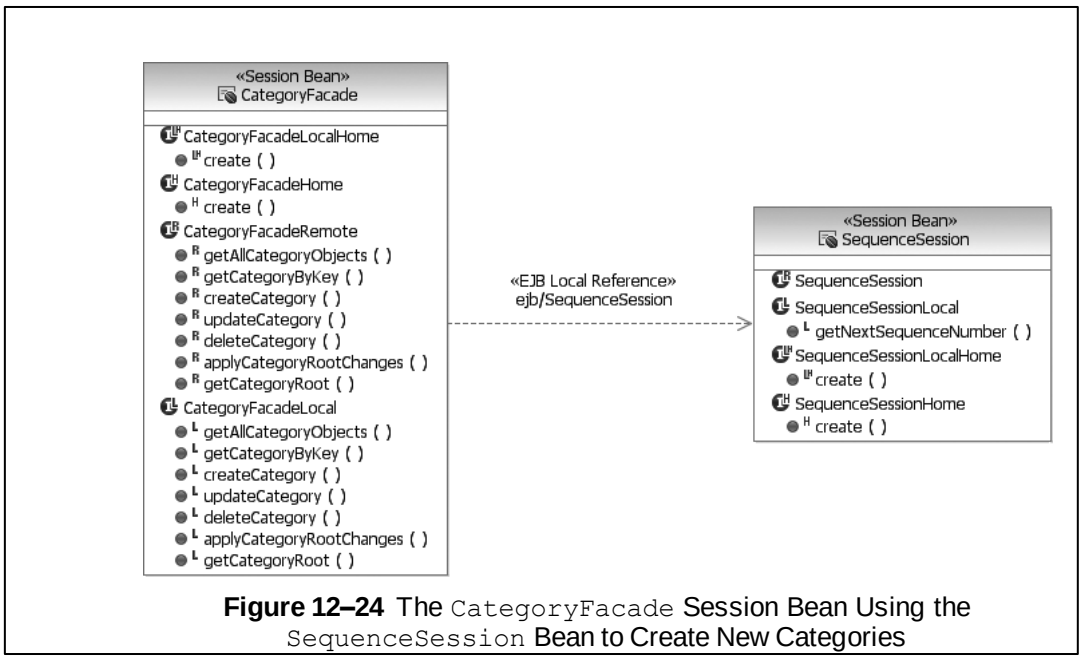

Figure 12–24 The `CategoryFacade` Session Bean Using the `SequenceSession` Bean to Create New Categories

3.3.3 Finders

Finder methods are crucial in EJB entity design as they enable searching for specific entity instances based on given criteria. These criteria can range from simple searches, like retrieving all vacation time request categories, to more complex ones, such as finding all employees whose leave requests in a certain category were rejected during a specific period. The key advantage of finder methods is that the filtering of matching entities occurs on the server, typically on the database server, which is the most efficient approach for handling large collections of entities. Without finder methods, retrieving all instances of an entity and then filtering them locally would be significantly less efficient.

The EJB Query Language (EJB QL) serves a similar purpose to SQL's SELECT statement but with some differences. For instance, the following EJB QL statement retrieves all Grant entity instances not linked to any restrictions.

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

Finders can also accept parameters. In the following example, the finder query retrieves all instances of DateExclusionRestriction that exclude the provided date. Parameters in the query are denoted by 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 defined in the main EJB configuration file `ejbjar.xml` and linked to the Java method declared in the home interface.

3.4 Controllers

Even in web applications, there are various approaches to managing business logic. One common approach is the model-view-controller (MVC) pattern. Currently, the Apache Struts framework is a popular choice for implementing MVC in J2EE web applications. Struts provides a runtime component and JSP tag libraries for easier coordination with the control framework. While MVC is a widely used pattern, there are many interpretations and implementations of it, with Struts being one of the first widely accepted implementations for Java-based web applications.

Upon closer examination of the system-level use cases, it becomes apparent that strict control in the presentation tier may not be necessary. In most scenarios, the required functionality is basic CRUD operations on entities. Our application primarily focuses on these operations rather than orchestrating complex business operations, which would be better suited for an MVC paradigm.

The architect of this system has made a deliberate choice not to adopt an overt MVC paradigm like Struts. While MVC is often considered critical for Web-centric applications, in this case, the architect has opted to rely on the inherent coordination and control provided by the Web pages themselves. This decision reflects a cautious approach, prioritizing simplicity and avoiding unnecessary complexity or dependencies that may not be immediately necessary for the successful release of the application.

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.

3.5 The Web Pages and the User Interface

Web page design involves two major concerns: the content and functionality of the pages themselves, and their layout and presentation. The former focuses on determining what conceptual pages are needed, their content, and how users navigate through them to achieve system goals. Web pages are typically implemented as JSP pages, containing a mix of presentation data and links to business processing beans. The latter concern, layout, focuses on organizing page elements for efficient user understanding and interaction.

Starting with the UX model, which maps to individual JSP pages, web designers can translate screen classes into page names, content, and navigation links. While pure HTML pages can be used for static content, JSP pages are preferred for dynamic content due to client-side state management requirements. Depending on the web tooling, the UX model can be transformed into a site layout or design model, with screens mapping to JSP pages and associations to navigation links.

In this application's J2EE architecture, JSF (JavaServer Faces) is utilized to simplify dynamic page construction and interaction with business logic processes. JSF provides custom tags embedded within HTML in JSP pages, enabling invocation of methods on business objects and extraction of data for rendering on client screens. Additionally, the Java Server Pages Standard Template Library (JSTL) offers useful tags for data manipulation in JSP pages.

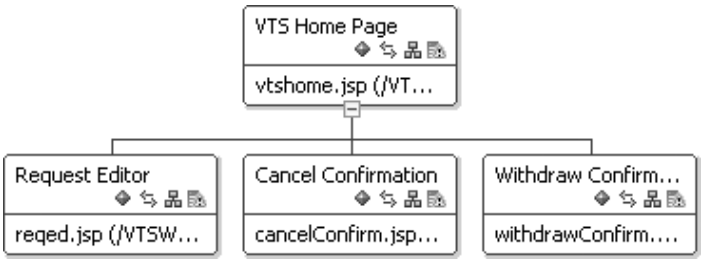

Figure 12–25 The Site Layout Represented in a Design Tool

3.5.1 Populating Dynamic Content

To populate a web page with dynamic content in the VTS home page, a connection is established between a Java session bean in the presentation tier and a session bean in the business tier. This connection, defined as a remote reference, enables potential separation of tiers onto different nodes.

The EmployeeSummarySession class, a session bean in the presentation tier, connects to the EmployeeFacade to create a summary of an Employee object's requests. It collects employee-specific information used in the VTS home page. The EmployeeSummary class and its public inner class RequestSummary are plain Java objects (POJOs) utilized directly in JSP pages via JSTL and JSF tags.

The JSP source code for the VTS home page incorporates these objects to display dynamic content.

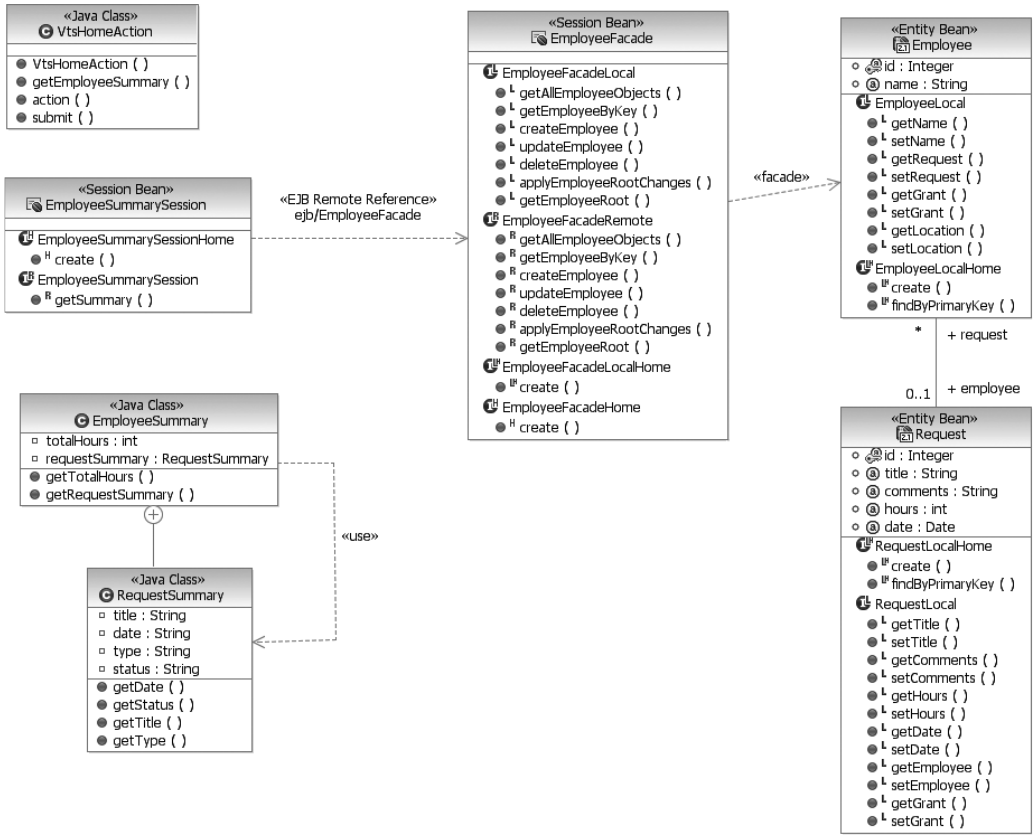

Figure 12–26 Design Elements in the Presentation and Business Tiers

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>
```

In the VTS home page, like other JSP pages, custom tags from various tag libraries are utilized, including core JSF tags, JSF HTML tags, and JSTL tags. These tags are imported and associated with prefix identifiers at the beginning of the JSP source.

A JSP tag is employed to reference a presentation tier bean, acting as a local controller for page actions and providing access to business state. This bean, implemented as a POJO, is referenced within the JSP context as "vtsHome."

For instance, the JSTL forEach tag iterates over the employeeSummary collection of the bean, containing instances of RequestSummary objects representing specific vacation time requests.

It's a best practice to isolate JSP code and logic from EJBs using a simple bean object paired with a specific page. This bean organizes and provides access to all the necessary data for the JSP, easing the burden on the creative team handling the page layout. They won't need to deal with the technical complexities of managing EJBs directly.

3.5.2 Invoking Business Logic

In web applications, business logic is primarily invoked during page transitions, triggered by user requests to view or navigate to the next page. The logic executed depends on the user's input or the current application state. For instance, if a user submits a form with missing information, the system may redirect them back to the form page with a warning message.

In JSF applications, navigation paths are defined in the faces-config.xml configuration file. This file also specifies managed beans, such as VtsHomeAction, which can be referenced in JSP pages. Navigation rules within faces-config.xml dictate the transitions between pages based on outcome values computed within managed beans. These outcomes determine which page to load as a response.

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 outlined in Listing 12–4 defines that activating a button or hyperlink component on reqed.jsp will lead to navigation from reqed.jsp to vtshome.jsp if the outcome is "success". Otherwise, the application returns to reqed.jsp.

Hyperlinks or form buttons are integrated into the originating page using JSF HTML tags. For instance, in the following command button example, clicking on the button triggers the submit() method of the vtsHomeAction instance. This method processes data and returns a logical outcome, which is then matched against navigation rules in the configuration file.

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

This design centralizes presentation logic in managed or backing beans executing in the Web server. These beans interact with remote EJBs in the business tier to perform tasks, potentially on different nodes within the system.

4 TRANSISTOIN

In the context of web-centric architectures, particularly those deployed on the internet, there are specific implementation and testing considerations beyond functionality and general performance. Web applications operate in a heterogeneous environment where client hardware and software configurations vary. To address these challenges during design, our VTS application intentionally avoids reliance on browser-specific technologies.

Web application development primarily involves server-side software. The majority of code executes on the server, with client-side concerns arising only when custom JavaScript, specialized applets, or client-side controls are utilized. In such cases, developers must consider the diverse client configurations the application may encounter.

Ensuring consistent performance and functionality across different browsers and client configurations is a significant challenge in web development, even when using officially supported technologies. Here are some key considerations and potential issues:

- **JavaScript**: Browsers may implement client-side scripting differently, with variations in support for features and interpretation of specifications. Proprietary extensions and differences in behavior can lead to inconsistencies.
- **Style Sheets**: Rendering of style sheets, including layout and font display, can vary depending on available fonts and screen sizes. While style sheets offer flexibility, they may not produce consistent results across all client configurations.
- **Frames**: Implementation of frames can be problematic, with issues related to scrolling, sizing, and targeting. Thorough testing is essential to ensure compatibility across targeted client configurations and usage scenarios.
- **Bandwidth**: As web development tools enable the incorporation of more complex features, pages may become heavier, leading to slower page transitions, especially for users with limited bandwidth or weaker client configurations. Designing and testing for low bandwidth and weak clients is crucial, particularly for applications deployed to a wide audience over the internet.

Addressing these challenges requires ongoing testing, adaptation, and consideration of evolving technologies and user needs.