Chapter 5

The J2EE patterns are a collection of J2EE-based solutions to common problems. They reflect the collective expertise and experience of Java architects at the Sun Java Center, gained from successfully executing numerous J2EE engagements. The Sun Java Center is Sun's consulting organization, focused on architecting Java technology-based solutions for customers. The Sun Java Center has been architecting solutions for the J2EE platform since its early days, focusing on achieving Quality of Service (QoS) qualities, such as scalability, availability, performance, securability, reliability, and flexibility.

These J2EE patterns describe typical problems encountered by enterprise application developers and provide solutions for these problems. We have formulated these solutions based on our ongoing work with numerous J2EE customers and on exchanges with other Java architects experiencing similar problems. The patterns capture the essence of these solutions, and they represent the solution refinement that takes place over the course of time and from collective experience. To put it another way, they extract the core issues of each problem, offering solutions that represent an applicable distillation of theory and practice.

Our work has focused on the J2EE area, especially regarding such J2EE components as Enterprise Java Beans (EJB), JavaServer Pages (JSP), and servlets. During our work with J2EE customers implementing the various components, we have come to recognize the common problems and difficult areas that may impede a good implementation. We've also developed effective best practices and approaches for using the J2EE components in combination.

The patterns presented here extract these “best practice” approaches and present them to you in a way that enables you to apply the patterns to your own particular application and to accommodate your own needs. The patterns clearly and simply express proven techniques. They make it easier for you to reuse successful designs and architectures. Simply put, you can use the patterns to design your J2EE system successfully and quickly.

What Is a Pattern?

In Chapter 1, we discussed how different experts define a pattern. We also discussed some of the peripheral issues around patterns including the benefits of using patterns. Here, we revisit this discussion in the context of the J2EE Pattern Catalog.

As discussed in Chapter 1, some experts define a pattern as a recurring **solution** to a **problem** in a **context**.

These terms—context, problem, and solution—deserve a bit of explanation. First, what is a context? A context is the environment, surroundings, situation, or interrelated conditions within which something exists. Second, what is a problem? A problem is an unsettled question, something that needs to be investigated and solved. Typically, the problem is constrained by the context in which it occurs. Finally, the solution refers to the answer to the problem in a context that helps resolve the issues.

So, if we have a solution to a problem in a context, is it a pattern? Not necessarily. The characteristic of recurrence also needs to be associated with the definition of a pattern. That is, a pattern is only useful if it can be applied repeatedly. Is that all? Perhaps not. As you can see, while the concept of a pattern is fairly simple, actually defining the term is more complex.

We point you to the references so that you can dig more deeply into the pattern history and learn about patterns in other areas. In our catalog, a pattern is described according to its main characteristics: **problem**, and **solution**, along with other important aspects, such as **forces** and **consequences**. The section describing the pattern template (see “Pattern Template” on page 129) explains these characteristics in more detail.

Identifying a Pattern

We have handled many J2EE projects at the Sun Java Center, and over time we have noticed that similar problems recur across these projects. We have also seen similar solutions emerge for these problems. While the implementation strategies varied, the overall solutions were quite similar. Let us discuss, in brief, our pattern identification process.

When we see a problem and solution recur, we try to identify and document its characteristics using the pattern template. At first, we consider these initial documents to be candidate patterns. However, we do not add candidate patterns to the pattern catalog until we are able to observe and document their usage multiple times on different projects. We also undertake the process of pattern mining by looking for patterns in implemented solutions.

As part of the pattern validation process, we use the **Rule of Three**, as it is known in the pattern community. This rule is a guide for transitioning a candidate pattern into the pattern catalog. According to this rule, a solution remains a candidate pattern until it has been verified in at least three different systems. Certainly, there is much room for interpretation with rules such as this, but they help provide a context for pattern identification.

Often, similar solutions may represent a single pattern. When deciding how to form the pattern, it is important to consider how to best communicate the solution. Sometimes, a separate name improves communication among developers. If so, then consider documenting two similar solutions as two different patterns. On the other hand, it might be better to communicate the solution by distilling the similar ideas into a pattern/strategy combination.

Patterns Versus Strategies

When we started documenting the J2EE patterns, we made the decision to document them at a relatively high level of abstraction. At the same time, each pattern includes various strategies that provide lower level implementation details. Through the strategies, each pattern documents a solution at multiple levels of abstraction. We could have documented some of these strategies as patterns in their own right; however, we feel that our current template structure most clearly communicates the relationship of the strategies to the higher level pattern structure in which they are included.

While we continue to have lively debates about converting these strategies to patterns, we have deferred these decisions for now, believing the current documentation to be clear. We have noted some of the issues with respect to the relationship of the strategies to the patterns:

- The patterns exist at a higher level of abstraction than the strategies.
- The patterns include the most recommended or most common implementations as strategies.

- Strategies provide an extensibility point for each pattern. Developers discover and invent new ways to implement the patterns, producing new strategies for well-known patterns.
- Strategies promote better communication by providing names for lower level aspects of a particular solution.

The Tiered Approach

Since this catalog describes patterns that help you build applications that run on the J2EE platform, and since a J2EE platform (and application) is a multitiered system, we view the system in terms of **tiers**. A tier is a logical partition of the separation of concerns in the system. Each tier is assigned its unique responsibility in the system. We view each tier as logically separated from one another. Each tier is loosely coupled with the adjacent tier. We represent the whole system as a stack of tiers. See Figure 5.1.

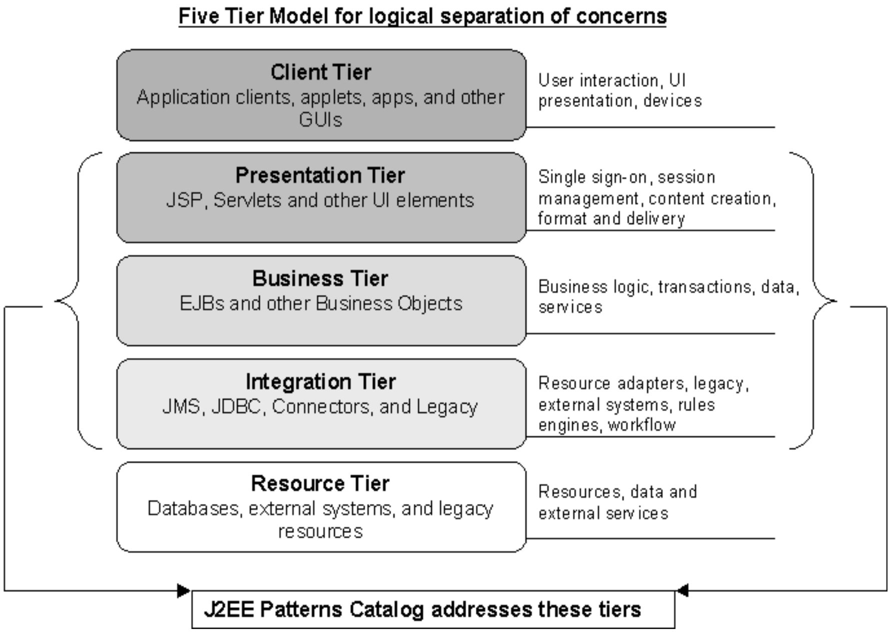
Five Tier Model for logical separation of concerns
Client Tier
Application clients, applets, apps, and other GUIs
User interaction, UI presentation, devices
Presentation Tier
JSP, Servlets and other UI elements
Single sign-on, session management, content creation, format and delivery
Business Tier
EJBs and other Business Objects
Business logic, transactions, data, services
Integration Tier
JMS, JDBC, Connectors, and Legacy
Resource adapters, legacy, external systems, rules engines, workflow
Resource Tier
Databases, external systems, and legacy resources
Resources, data and external services
J2EE Patterns Catalog addresses these tiers

Figure 5.1 Tiered approach

Client Tier

This tier represents all device or system clients accessing the system or the application. A client can be a Web browser, a Java or other application, a Java applet, a WAP phone, a network application, or some device introduced in the future. It could even be a batch process.

Presentation Tier

This tier encapsulates all presentation logic required to service the clients that access the system. The presentation tier intercepts the client requests, provides single sign-on, conducts session management, controls access to business services, constructs the responses, and delivers the responses to the client. Servlets and JSP reside in this tier. Note that servlets and JSP are not themselves UI elements, but they produce UI elements.

Business Tier

This tier provides the business services required by the application clients. The tier contains the business data and business logic. Typically, most business processing for the application is centralized into this tier. It is possible that, due to legacy systems, some business processing may occur in the resource tier. Enterprise bean components are the usual and preferred solution for implementing the business objects in the business tier.

Integration Tier

This tier is responsible for communicating with external resources and systems such as data stores and legacy applications. The business tier is coupled with the integration tier whenever the business objects require data or services that reside in the resource tier. The components in this tier can use JDBC, J2EE connector technology, or some proprietary middleware to work with the resource tier.

Resource Tier

This is the tier that contains the business data and external resources such as mainframes and legacy systems, business-to-business (B2B) integration systems, and services such as credit card authorization.

J2EE Patterns

We used the tiered approach to divide the J2EE patterns according to functionality, and our pattern catalog follows this approach. The presentation tier patterns contain the patterns related to servlets and JSP technology. The business tier patterns contain the patterns related to the EJB technology. The integration tier patterns contain the patterns related to JMS and JDBC. See Figure 5.2 on page 132.

Presentation Tier Patterns

Table 5-1 lists the presentation tier patterns, along with a brief description of each pattern.

Table 5-1 Presentation Tier Patterns

Pattern Name	***Synopsis***
Intercepting Filter (144)	Facilitates preprocessing and post-processing of a request.
Front Controller (166)	Provides a centralized controller for managing the handling of a request.
Context Object (181)	Encapsulates state in a protocol-independent way to be shared throughout your application.
Application Controller (205)	Centralizes and modularizes action and view management.
View Helper (240)	Encapsulates logic that is not related to presentation formatting into Helper components.
Composite View (262)	Creates an aggregate View from atomic subcomponents.
Service to Worker (276)	Combines a Dispatcher component with the *Front Controller* (166) and *View Helper* (240) patterns.
Dispatcher View (288)	Combines a Dispatcher component with the *Front Controller* (166) and *View Helper* (240) patterns, deferring many activities to View processing.

Business Tier Patterns

Table 5-2 lists the business tier patterns, along with a brief synopsis of each pattern.

Table 5-2 Business Tier Patterns

Pattern Name	*Synopsis*
Business Delegate (302)	Encapsulates access to a business service.
Service Locator (315)	Encapsulates service and component lookups.
Session Façade (341)	Encapsulates business-tier components and exposes a coarse-grained service to remote clients.
Application Service (357)	Centralizes and aggregates behavior to provide a uniform service layer.
Business Object (374)	Separates business data and logic using an object model.
Composite Entity (391)	Implements persistent *Business Objects* (374) using local entity beans and POJOs.
Transfer Object (415)	Carries data across a tier.
Transfer Object Assembler (433)	Assembles a composite transfer object from multiple data sources.
Value List Handler (444)	Handles the search, caches the results, and provide the ability to traverse and select items from the results.

Integration Tier Patterns

Table 5-3 lists the integration tier patterns and a brief description of each pattern.

Table 5-3 Integration Tier Patterns

Pattern Name	*Synopsis*
Data Access Object (462)	Abstracts and encapsulates access to persistent store.

Table 5-3 Integration Tier Patterns (continued)

Pattern Name	*Synopsis*
Service Activator (496)	Receives messages and invokes processing asynchronously.
Domain Store (516)	Provides a transparent persistence mechanism for business objects.
Web Service Broker (557)	Exposes one or more services using XML and web protocols

Guide to the Catalog

To help you effectively understand and use the J2EE patterns in the catalog, we suggest that you familiarize yourself with this section before reading the individual patterns. Here we introduce the pattern terminology and explain our use of the Unified Modeling Language (UML), stereotypes, and the pattern template. In short, we explain how to use these patterns. We also provide a high-level roadmap to the patterns in the catalog.

Terminology

Players in the enterprise computing area, and particularly establishments using Java-based systems, have incorporated a number of terms and acronyms into their language. While many readers are familiar with these terms, sometimes their use varies from one setting to another. To avoid misunderstandings and to keep things consistent, we define in Table 5-4 how we use these terms and acronyms.

Table 5-4 Terminology

Term	*Description/Definition*	*Used In*
BMP	Bean-managed persistence: a strategy for entity beans where the bean developer implements the persistence logic for entity beans.	Business tier patterns

Table 5-4 Terminology (continued)

Term	*Description/Definition*	*Used In*
CMP	Container-managed persistence: a strategy for entity beans where the container services transparently manage the persistence of entity beans.	Business tier patterns
Composite	A complex object that holds other objects. Also related to the Composite pattern described in the GoF book. (See GoF later in this table.)	*Composite View* (262), *Composite Entity* (391)
Controller	Interacts with a client, controlling and managing the handling of each request.	Presentation and business tier patterns
Data Access Object	An object that encapsulates and abstracts access to data from a persistent store or an external system.	Business and integration tier patterns
Delegate	A stand-in, or surrogate, object for another component; an intermediate layer. A Delegate has qualities of a proxy and façade.	*Business Delegate* (302) and many other patterns
Dependent Object	An object that does not exist by itself and whose lifecycle is managed by another object.	*Business Objects* (374) and *Composite Entity* (391)
Dispatcher	Some of the responsibilities of a Controller include managing the choice of and dispatching to an appropriate View. This behavior may be partitioned into a separate component, referred to as a Dispatcher.	*Dispatcher View* (288), *Service to Worker* (276)

Table 5-4 Terminology (continued)

Term	*Description/Definition*	*Used In*
Enterprise Bean	Refers to an Enterprise JavaBean component; can be a session or entity bean instance. When this term is used, it means that the bean instance can be either an entity or a session bean.	Many places in this literature
Façade	A pattern for hiding underlying complexities; described in the GoF book.	*Session Façade* pattern
Factory (Abstract Factory or Factory Method)	Patterns described in the GoF book for creating objects or families of objects.	Business tier patterns: *Data Access Object* (462)
Iterator	A pattern to provide accessors to underlying collection facilities; described in the GoF book.	*Value List Handler (444)*
GoF	Gang of Four—refers to the authors of the popular design patterns book, *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. [GoF]	Many places in this literature
Helper	Responsible for helping the Controller and/or View. For example, the Controller and View may delegate the following to a Helper: content retrieval, validation, storing the model or adapting it for use by the display.	Presentation tier patterns, *Business Delegate* (302)
Independent Object	An object that can exist by itself and may manage the lifecycles of its dependent objects.	*Composite Entity* (391) pattern

Table 5-4 Terminology (continued)

Term	*Description/Definition*	*Used In*
Model	A physical or logical representation of the system or its subsystem.	Presentation and business tier patterns
Persistent Store	Represents persistent storage systems such as RDBMSs, ODBMSs, file systems, and so forth.	Business and integration tier patterns
Proxy	A pattern to provide a placeholder for another object to control access to it; described in the GoF book.	Many places in this literature
Scriptlet	Application logic embedded directly within a JSP.	Presentation tier patterns
Session Bean	Refers to a stateless or stateful session bean. May also refer collectively to the session bean's home, remote object, and bean implementation.	Business tier patterns
Singleton	A pattern that provides a single instance of an object, as described in the GoF book.	Many places in this literature
Template	Template text refers to the literal text encapsulated within a JSP View. Additionally, a template may refer to a specific layout of components in a display.	Presentation tier patterns
Transfer Object	A serializable POJO that is used to carry data from one object/tier to another. It does not contain any business methods.	Business tier patterns

Table 5-4 Terminology (continued)

Term	*Description/Definition*	*Used In*
View	The View manages the graphics and text that make up the display. It interacts with Helpers to get data values with which to populate the display. Additionally, it may delegate activities, such as content retrieval, to its Helpers.	Presentation tier patterns

Use of UML

We have used UML extensively in the pattern catalog, particularly as follows:

- **Class diagrams** – We use the class diagrams to show the structure of the pattern solution and the structure of the implementation strategies. This provides the static view of the solution.
- **Sequence (or Interaction) diagrams** – We use these diagrams to show the interactions between different participants in a solution or a strategy. This provides the dynamic view of the solution.
- **Stereotypes** – We use stereotypes to indicate different types of objects and roles in the class and interaction diagrams. The list of stereotypes and their meanings is included in Table 5-5.

Each pattern in the pattern catalog includes a class diagram that shows the structure of the solution and a sequence diagram that shows the interactions for the pattern. In addition, patterns with multiple strategies use class and sequence diagrams to explain each strategy.

To learn more about UML, please see the Bibliography.

UML Stereotypes

While reading the patterns and their diagrams, you will encounter certain stereotypes. Stereotypes are terms coined or used by designers and architects. We created and used these stereotypes to present the diagrams in a concise and easy to understand manner. Note that some of the stereotypes relate to the terminology explained in the previous section. In addition to these stereotypes, we also use pattern names and the main roles of a pattern as the stereotypes when it helps in explaining a pattern and its strategies.

Table 5-5 UML Stereotypes

Stereotype	*Meaning*
EJB	Represents an enterprise bean component; associated with a business object. This is a role that is usually fulfilled by a session or entity bean.
SessionEJB	Represents a session bean as a whole without specifying the session bean remote interface, home interface, or the bean implementation.
EntityEJB	Represents an entity bean as a whole without specifying the entity bean remote interface, home interface, the bean implementation, or the primary key.
View	A View represents and displays information to the client.
JSP	A JavaServer Page; a View is typically implemented as a JSP.
Servlet	A Java servlet; a Controller is typically implemented as a servlet.
Singleton	A class that has a single instance in accordance with the Singleton pattern.
Custom Tag	JSP custom tags are used to implement Helper objects, as are JavaBeans. A Helper is responsible for such activities as gathering data required by the View and for adapting this data model for use by the View. Helpers can service requests for data from the View by simply providing access to the raw data or by formatting the data as Web content.

Pattern Template

The J2EE patterns are all structured according to a defined pattern template. The pattern template consists of sections presenting various attributes for a given pattern. You'll also notice that we've tried to give each J2EE pattern a descriptive pattern name. While it is difficult to fully encompass a single pattern in its name, the pattern names are intended to provide sufficient insight into the function of the pattern. Just as with names in real life, those assigned to patterns affect how the reader will interpret and eventually use that pattern.

We have adopted a pattern template that consists of the following sections:

- *Problem:* Describes the design issues faced by the developer.
- *Forces:* Lists the reasons and motivations that affect the problem and the solution. The list of forces highlights the reasons why one might choose to use the pattern and provides a justification for using the pattern.
- *Solution:* Describes the solution approach briefly and the solution elements in detail. The solution section contains two subsections:
    - *Structure:* Uses UML class diagrams to show the basic structure of the solution. The UML Sequence diagrams in this section present the dynamic mechanisms of the solution. There is a detailed explanation of the participants and collaborations.
    - *Strategies:* Describes different ways a pattern may be implemented. Please see “Patterns Versus Strategies” on page 119 to gain a better understanding of strategies. Where a strategy can be demonstated using code, we include a code snippet in this section. If the code is more elaborate and lengthier than a snippet, we include it in the “Sample Code” section of the pattern template.
- *Consequences:* Here we describe the pattern trade-offs. Generally, this section focuses on the results of using a particular pattern or its strategy, and notes the pros and cons that may result from the application of the pattern.
- *Sample Code:* This section includes example implementations and code listings for the patterns and the strategies. This section is rendered optional if code samples can be adequately included with the discussion in the “Strategies” section.
- *Related Patterns:* This section lists other relevant patterns in the J2EE Pattern Catalog or from other external resources, such as the GoF design patterns. For each related pattern, there is a brief description of its relationship to the pattern being described.

J2EE Pattern Relationships

A recent focus group of architects and designers raised a major concern: There seems to be a lack of understanding of how to apply patterns in combination to form larger solutions. We address this problem with a high-level visual of the patterns and their relationships. This diagram is called the J2EE Pattern Relationships Diagram and is shown in Figure 5.2. In the Epilogue, “Web Worker

Micro-Architecture”, we explore example use cases to demonstrate how many patterns come together to form a patterns framework to realize a use case.

Individual patterns offer their context, problem, and solution when addressing a particular need. However, it is important to step back and grasp the big picture to put the patterns to their best use. This grasping the big picture results in better application of the patterns in a J2EE application.

Reiterating Christopher Alexander’s quote from Chapter 1, a pattern does not exist in isolation and needs the support of other patterns to bring meaning and usefulness. Virtually every pattern in the catalog has a relationship to other patterns. Understanding these relationships when designing and architecting a solution helps in the following ways:

- Enables you to consider what other new problems may be introduced when you consider applying a pattern to solve your problem. This is the domino effect: What new problems are introduced when a particular pattern is introduced into the architecture? It is critical to identify these conflicts before coding begins.
- Enables you to revisit the pattern relationships to determine alternate solutions. After possible problems are identified, revisit the pattern relationships and collect alternate solutions. Perhaps the new problems can be addressed by selecting a different pattern or by using another pattern in combination with the one you have already chosen.

Figure 5.2 on page 132 shows the relationships among the patterns.

Intercepting Filter (144) intercepts incoming requests and outgoing responses and applies a filter. These filters may be added and removed in a declarative manner, allowing them to be applied unobtrusively in a variety of combinations. After this preprocessing and/or post-processing is complete, the final filter in the group vectors control to the original target object. For an incoming request, this is often a *Front Controller* (166), but may be a *View* (240).

Front Controller (166) is a container to hold the common processing logic that occurs within the presentation tier and that may otherwise be erroneously placed in a View. A controller handles requests and manages content retrieval, security, view management, and navigation, delegating to a Dispatcher component to dispatch to a View.

Application Controller (205) centralizes control, retrieval, and invocation of view and command processing. While a *Front Controller* (166) acts as a centralized access point and controller for incoming requests, the *Application Controller* (205) is responsible for identifying and invoking commands, and for identifying and dispatching to views.

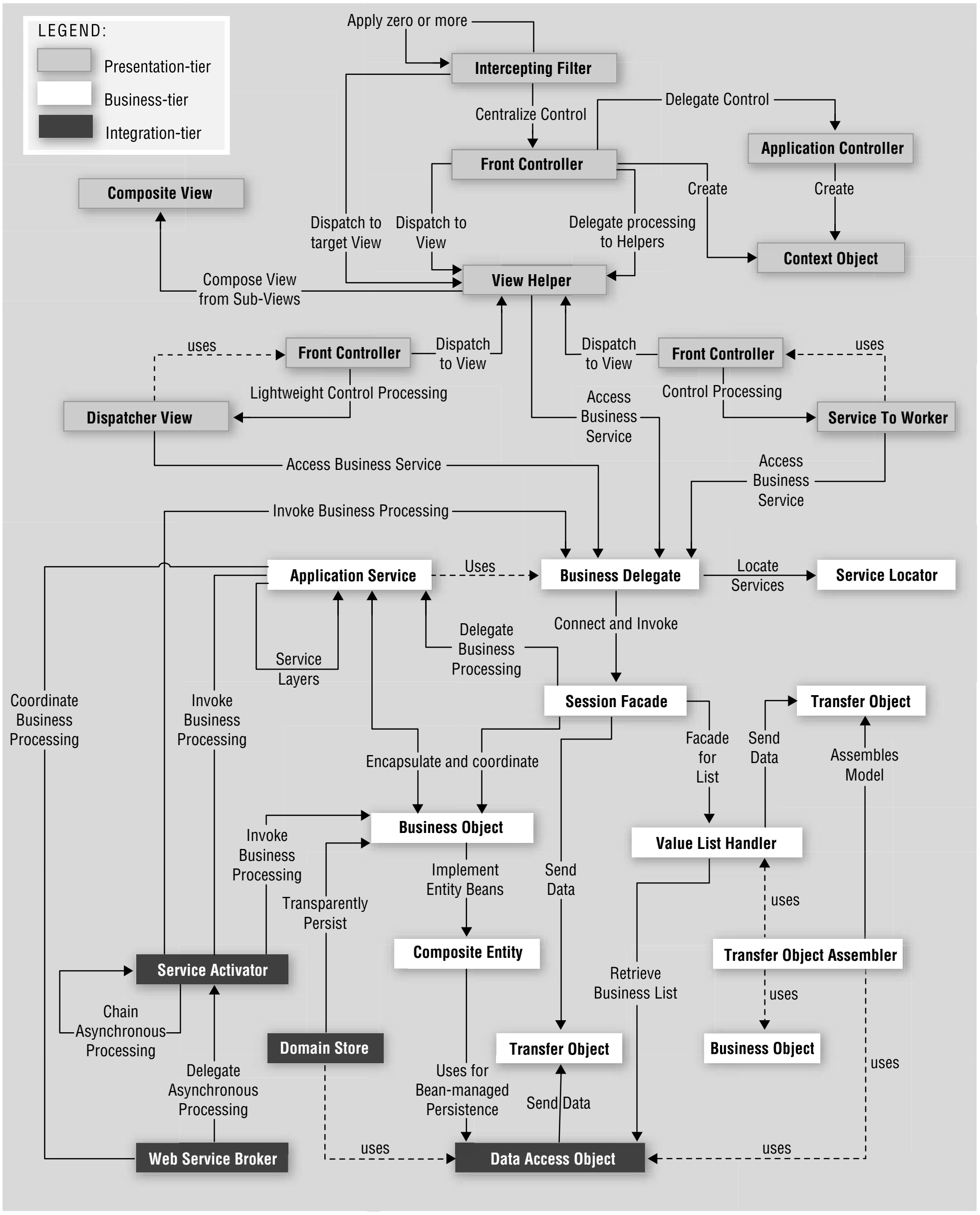

Figure 5.2 J2EE Pattern Relationships

Context Object (181) encapsulates state in a protocol-independent way to be shared throughout your application. Using *Context Object* (181) makes testing easier, facilitating a more generic test environment with reduced dependence upon a specific container.

View Helper (240) encourages the separation of formatting-related code from other business logic. It suggests using Helper components to encapsulate logic relating to initiating content retrieval, validation, and adapting and formatting the model. The View component is then left to encapsulate the presentation formatting. Helper components typically delegate to the business services via a *Business Delegate* (302) or an *Application Service* (357), while a View may be composed of multiple subcomponents to create its template.

Composite View (262) suggests composing a View from numerous atomic pieces. Multiple smaller views, both static and dynamic, are pieced together to create a single template. The *Service to Worker* (276) and *Dispatcher View* (288) patterns represent a common combination of other patterns from the catalog. The two patterns share a common structure, consisting of a controller working with a Dispatcher, Views, and Helpers. *Service to Worker* (276) and *Dispatcher View* (288) have similar participant roles, but differ in the division of labor among those roles. Unlike *Service to Worker* (276), *Dispatcher View* (288) defers business processing until view processing has been performed.

Business Delegate (302) reduces coupling between remote tiers and provides an entry point for accessing remote services in the business tier. A *Business Delegate* (302) might also cache data as necessary to improve performance. A *Business Delegate* (302) encapsulates a *Session Façade* (341) and maintains a one-to-one relationship with that *Session Façade* (341). An *Application Service* (357) uses a *Business Delegate* (302) to invoke a *Session Façade* (341).

Service Locator (315) encapsulates the implementation mechanisms for looking up business service components. A *Business Delegate* (302) uses a *Service Locator* (315) to connect to a *Session Façade* (341). Other clients that need to locate and connect to *Session Façade* (341), other business-tier services, and web services can use a *Service Locator* (315).

Session Façade (341) provides coarse-grained services to the clients by hiding the complexities of the business service interactions. A *Session Façade* (341) might invoke several *Application Service* (357) implementations or *Business Objects* (374). A *Session Façade* (341) can also encapsulate a *Value List Handler* (444).

Application Service (357) centralizes and aggregates behavior to provide a uniform service layer to the business tier services. An *Application Service* (357) might interact with other services or *Business Objects* (374). An *Application Ser-*

vice (357) can invoke other *Application Services* (357*)* and thus create a layer of services in your application.

Business Object (374) implements your conceptual domain model using an object model. *Business Objects* (374) separate business data and logic into a separate layer in your application. *Business Objects* (374) typically represent persistent objects and can be transparently persisted using *Domain Store* (516).

Composite Entity (391) implements a *Business Object* (374) using local entity beans and POJOs. When implemented with bean-managed persistence, a *Composite Entity* (391) uses *Data Access Objects* (462) to facilitate persistence.

The *Transfer Object* (415) pattern provides the best techniques and strategies to exchange data across tiers (that is, across system boundaries) to reduce the network overhead by minimizing the number of calls to get data from another tier.

The *Transfer Object Assembler* constructs a composite *Transfer Object* (415) from various sources. These sources could be EJB components, *Data Access Objects* (462), or other arbitrary Java objects. This pattern is most useful when the client needs to obtain data for the application model or part of the model.

The *Value List Handler* (444) uses the GoF iterator pattern to provide query execution and processing services. The *Value List Handler* (444) caches the results of the query execution and return subsets of the result to the clients as requested. By using this pattern, it is possible to avoid overheads associated with finding large numbers of entity beans. The *Value List Handler* (444) uses a *Data Access Object* (462) to execute a query and fetch the results from a persistent store.

Data Access Object (462) enables loose coupling between the business and resource tiers. *Data Access Object* (462) encapsulates all the data access logic to create, retrieve, delete, and update data from a persistent store. *Data Access Object* (462) uses *Transfer Object* (415) to send and receive data.

Service Activator (496) enables asynchronous processing in your enterprise applications using JMS. A *Service Activator* (496) can invoke *Application Service* (357), *Session Façade* (341) or *Business Objects* (374). You can also use several *Service Activators* (496*)* to provide parallel asynchronous processing for long running tasks.

Domain Store (516) provides a powerful mechanism to implement transparent persistence for your object model. It combines and links several other patterns including *Data Access Objects* (462). *Web Service Broker* (557) exposes and brokers one or more services in your application to external clients as a web service using XML and standard web protocols. A *Web Service Broker* (557) can interact with *Application Service* (357) and *Session Façade* (341). A *Web Service Broker* (557) uses one or more *Service Activators* (496*)* to perform asynchronous processing of a request.

Relationship to Known Patterns

There is a wealth of software pattern documentation available today. The patterns in these different books are at various levels of abstraction. There are architecture patterns, design patterns, analysis patterns, and programming patterns. The most popular and influential of these books is *Design Patterns: Elements of Reusable Object-Oriented Software,* [GoF] better known as the Gang of Four, or GoF book. The patterns in the GoF book describe expert solutions for object design. We also reference patterns from *Patterns of Enterprise Application Architecture* [PEAA], by Martin Fowler.

Our pattern catalog includes patterns that describe the structure of an application and others that describe design elements. The unifying theme of the pattern catalog is its support of the J2EE platform. In some cases, the patterns in the catalog are based on or related to an existing pattern in the literature. In these cases, we communicate this relationship by referencing the existing pattern in the name of the J2EE pattern and/or including a reference and citation in the *Related Patterns* section at the end of each pattern description. For example, some patterns are based on GoF patterns but are considered in a J2EE context. In those cases, the J2EE pattern name includes the GoF pattern name as well as a reference to the GoF pattern in the related patterns section.

Patterns Roadmap

Here we present a list of common requirements that architects encounter when creating solutions with the J2EE. We present the requirement or motivation in a brief statement, followed by a list of one or more patterns addressing that requirement. While this requirements list is not exhaustive, we hope that it helps you to quickly identify the relevant patterns based on your needs.

Table 5-1 shows the functions typically handled by the presentation tier patterns and indicates which pattern provides a solution.

Table 5-1 Presentation Tier Patterns

If you are looking for this	*Find it here*
Preprocessing or postprocessing of your requests	**Pattern** *Intercepting Filter* (144)
Adding logging, debugging, or some other behavior to be completed for each request	**Pattern** *Front Controller* (166) **Pattern** *Intercepting Filter* (144)

Table 5-1 Presentation Tier Patterns (continued)

If you are looking for this	*Find it here*
Centralizing control for request handling	**Pattern** *Front Controller* (166) **Pattern** *Intercepting Filter* (144) **Pattern** *Application Controller* (205)
Creating a generic command interface or context object for reducing coupling between control components and helper components	**Pattern** *Front Controller* (166) **Pattern** *Application Controller* (205) **Pattern** *Context Object* (181)
Whether to implement your Controller as a servlet or JSP	**Pattern** *Front Controller* (166)
Creating a View from numerous sub-Views	**Pattern** *Composite View* (262)
Whether to implement your View as a servlet or JSP	**Pattern** *View Helper* (240)
How to partition your View and Model	**Pattern** *View Helper* (240)
Where to encapsulate your presentation-related data formatting logic	**Pattern** *View Helper* (240)
Whether to implement your Helper components as JavaBeans or Custom tags	**Pattern** *View Helper* (240)
Combining multiple presentation patterns	**Pattern** *Intercepting Filter* (144) **Pattern** *Dispatcher View* (288)
Where to encapsulate View Management and Navigation logic, which involves choosing a View and dispatching to it	**Pattern** *Service to Worker* (276) **Pattern** *Dispatcher View* (288)
Where to store session state	**Design** "Session State on the Client" (20) **Design** "Session State in the Presentation Tier" (21) **Design** "Storing State on the Business Tier" (48)

Table 5-1 Presentation Tier Patterns (continued)

If you are looking for this	*Find it here*
Controlling client access to a certain View or sub-View	**Design** "Controlling Client Access" (22) **Refactoring** "Hide Resources From a Client" (88)
Controlling the flow of requests into the application	**Design** "Duplicate Form Submissions" (27) **Design** "Introduce Synchronizer Token" (67)
Controlling duplicate form submissions	**Design** "Duplicate Form Submissions" (27) **Refactoring** "Introduce Synchronizer Token" (67)
Design issues using JSP standard property auto-population mechanism via <jsp:setProperty>	**Design** "Helper Properties—Integrity and Consistency" (30)
Reducing coupling between presentation tier and business tier	**Refactoring** "Hide Presentation Tier-Specific Details From the Business Tier" (80) **Refactoring** "Introduce Business Delegate" (94)
Partitioning data access code	**Refactoring** "Separate Data Access Code" (102)

Table 5-2 shows the functions handled by the business tier patterns and indicates where you can find the particular pattern or patterns that may provide solutions.

Table 5-2 Business Tier Patterns

If you are looking for this	*Find it here*
Minimize coupling between presentation and business tiers	**Pattern** *Business Delegate* (302)
Cache business services for clients	**Pattern** *Business Delegate* (302)

Table 5-2 Business Tier Patterns (continued)

If you are looking for this	*Find it here*
Hide implementation details of business service lookup/creation/access	**Pattern** *Business Delegate* (302) **Pattern** *Service Locator* (315)
Isolate vendor and technology dependencies for services lookup	**Pattern** *Service Locator* (315)
Provide uniform method for business service lookup and creation	**Pattern** *Service Locator* (315)
Hide the complexity and dependencies for enterprise bean and JMS component lookup	**Pattern** *Service Locator* (315)
Transfer data between business objects and clients across tiers	**Pattern** *Transfer Object* (415)
Provide simpler uniform interface to remote clients	**Pattern** *Business Delegate* (302) **Pattern** *Session Façade* (341) **Pattern** *Application Service* (357)
Reduce remote method invocations by providing coarse-grained method access to business tier components	**Pattern** *Session Façade* (341)
Manage relationships between enterprise bean components and hide the complexity of interactions	**Pattern** *Session Façade* (341)
Protect the business tier components from direct exposure to clients	**Pattern** *Session Façade* (341) **Pattern** *Application Service* (357)
Provide uniform boundary access to business tier components	**Pattern** *Session Façade* (341) **Pattern** *Application Service* (357)
Implement complex conceptual domain model using objects	**Pattern** *Business Object* (374)

Table 5-2 Business Tier Patterns (continued)

If you are looking for this	*Find it here*
Identify coarse-grained objects and dependent objects for business objects and entity bean design	**Pattern** *Business Object* (374) **Pattern** *Composite Entity* (391)
Design for coarse-grained entity beans	**Pattern** *Composite Entity* (391)
Reduce or eliminate the entity bean clients' dependency on the database schema	**Pattern** *Composite Entity* (391)
Reduce or eliminate entity bean to entity bean remote relationships	**Pattern** *Composite Entity* (391)
Reduce number of entity beans and improve manageability	**Pattern** *Composite Entity* (391)
Obtain the application data model for the application from various business tier components	**Pattern** *Transfer Object Assembler* (433)
On the fly construction of the application data model	**Pattern** *Transfer Object Assembler* (433)
Hide the complexity of data model construction from the clients	**Pattern** *Transfer Object Assembler* (433)
Provide business tier query and results list processing facility	**Pattern** *Value List Handler* (444)
Minimize the overhead of using enterprise bean finder methods	**Pattern** *Value List Handler* (444)
Provide query-results caching for clients on the server side with forward and backward navigation	**Pattern** *Value List Handler* (444)
Trade-offs between using stateful and stateless session beans	**Design** "Session Bean—Stateless Versus Stateful" (46)
Provide protection to entity beans from direct client access	**Refactoring** "Wrap Entities With Session" (92)
Encapsulate business services to hide the implementation details of the business tier	**Refactoring** "Introduce Business Delegate" (94)

Table 5-2 Business Tier Patterns (continued)

If you are looking for this	*Find it here*
Coding business logic in entity beans	**Design** "Business Logic in Entity Beans" (50) **Refactoring** "Move Business Logic to Session" (100)
Provide session beans as coarse-grained business services	**Refactoring** "Merge Session Beans" (96) **Refactoring** "Wrap Entities With Session" (92)
Minimize and/or eliminate network and container overhead due to entity-bean-to-entity-bean communication	**Refactoring** "Reduce Inter-Entity Bean Communication" (98)
Partition data access code	**Refactoring** "Separate Data Access Code" (102)

Table 5-3 shows the functions typically handled by the presentation tier patterns and indicates which pattern provides a solution.

Table 5-3 Integration Tier Patterns

If you are looking for this	*Find it here*
Minimize coupling between business and resource tiers	**Pattern** *Data Access Object* (462)
Centralize access to resource tiers	**Pattern** *Data Access Object* (462)
Minimize complexity of resource access in business tier components	**Pattern** *Data Access Object* (462)
Provide asynchronous processing for enterprise applications	**Pattern** *Service Activator* (496)
Send an asynchronous request to a business service	**Pattern** *Service Activator* (496)
Asynchronously process a request as a set of parallel tasks	**Pattern** *Service Activator* (496)
Transparently persist an object model	**Pattern** *Domain Store* (516)

Table 5-3 Integration Tier Patterns (continued)

If you are looking for this	***Find it here***
Implement a custom persistence framework	**Pattern** *Domain Store* (516)
Expose a web service using XML and standard Internet protocol	**Pattern** *Web Service Broker* (557)
Aggregate and broker existing services as web services	**Pattern** *Web Service Broker* (557)

Summary

So far, we have seen the basic concepts behind the J2EE patterns, understood the tiers for pattern categorization, explored the relationships between different patterns, and taken a look at the roadmap to help guide you to a particular pattern. In the following chapters, we present the patterns individually.

There are three chapters, for presentation, business, and integration tiers. Refer to the appropriate chapter to find the pattern you're interested in.