

Introducing the Architectural Tiers

Objectives

Upon completion of this module, you should be able to:

- Describe the concepts of the Client and Presentation tiers
- Describe the concepts of the Business tier
- Describe the concepts of the Resource and Integration tiers
- Describe the concepts of the Solution model

Additional Resources



Additional resources – The following references provide additional information on the topics described in this module:

- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. West Sussex, England: John Wiley & Sons, Ltd., 1996.
- Norman, Donald A. *The Design of Everyday Things*. New York: Currency/Doubleday, 1988.
- Sun Microsystems, Inc. *Java Look and Feel Design Guidelines*. Reading: Addison Wesley, 1999.
- Sun Microsystems, Inc. *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*.
[http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html], accessed 6 October 2002.
- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Grosso, William. *Java RMI*. Sebastopol: O'Reilly & Associates, Inc. 2002.
- The Object Management Group. "Unified Modeling Language (UML), Version 2.2"
[<http://www.omg.org/technology/documents/formal/uml.html>].
- Ambler, Scott. "Mapping Objects to Relational Databases,"
[<http://www.ambysoft.com/mappingObjects.pdf>].
accessed 2 October 2002.
- Booch, Grady. *Object-Oriented Analysis and Design with Applications (2nd ed)*. The Benjamin/Cummins Publishing Company, Inc., Redwood City, 1994.
- Gultzan, Peter, Trudy Pelzer. *SQL-99 Complete, Really*. Lawrence: R&D Books, 1999.

Process Map

This module describes the creation of the Architecture tiers. Figure 12-1 shows the activities and artifacts discussed in this module.

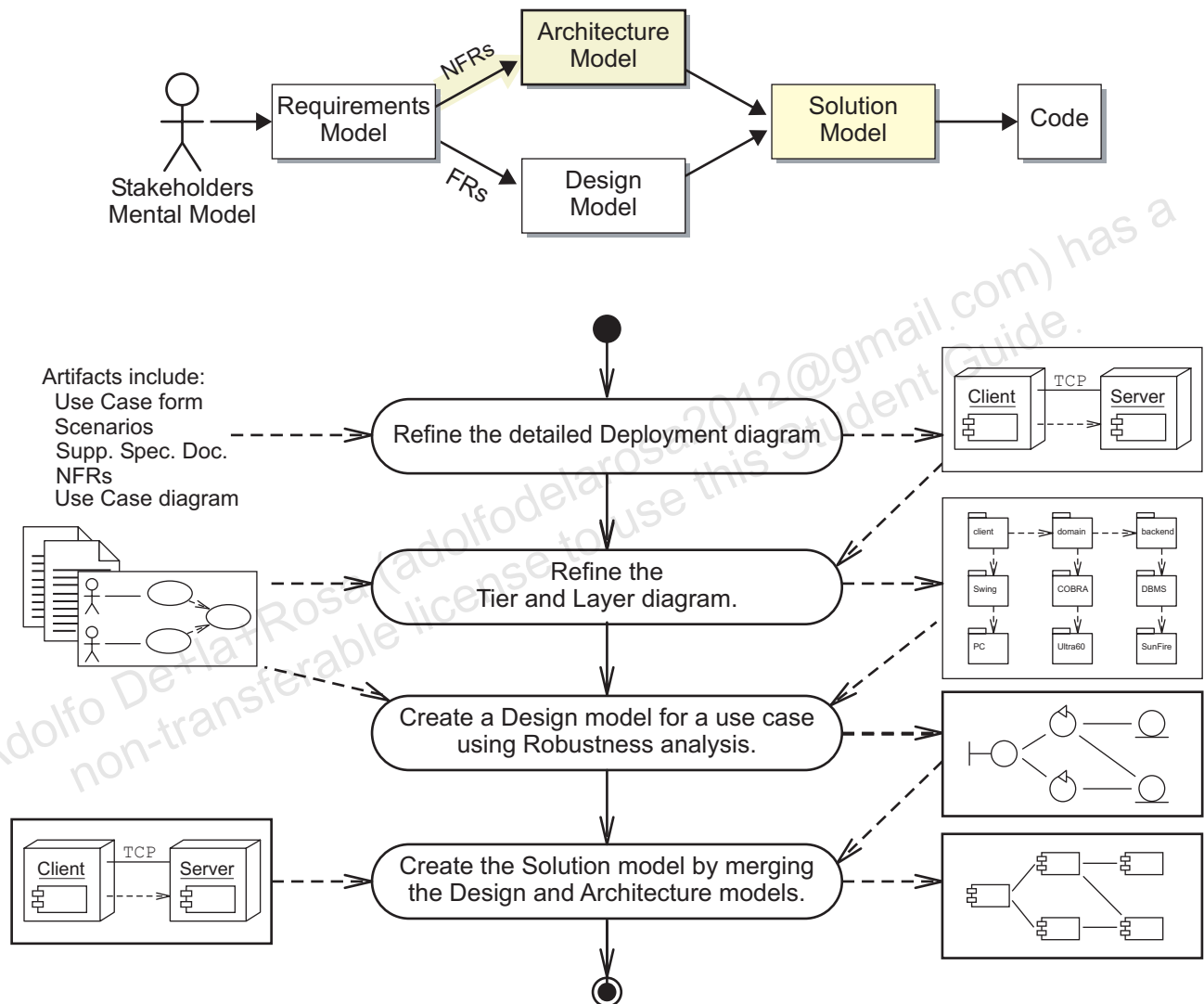


Figure 12-1 Architecture Tiers Process Map

Introducing the Client and Presentation Tiers

The Client and Presentation tiers primarily contain:

- Controller components:
 - Control the input from the boundary
 - Perform input sanity checking
Input sanity checking will ensure that the data submitted is plausible. For example, it can check that the dates entered are valid.
 - Call business logic methods
 - Dispatch view components
- View components:
 - Retrieve data required by the view
 - Prepare the view in a format suitable for the recipient
 - Add client-side sanity checking (optional)
Client-side sanity checking in a Web client could be performed by the browser by using JavaScript™.

The whole or part of the boundary may be replaced by alternative components in order to communicate with different actors (Human or Device) without the necessity to change the Business tiers. For example, the Hotel System would have different boundary components for:

- Local application access (for example, Swing)
- Web access, which may be partially substituted for:
 - Mobile phone browser
 - Standard browser
- Web service access to the Travel Agent System

Boundary Interface Technologies

The primary Boundary Interface types are:

- Graphical user interface (GUI)
A GUI is any system that provides a visual experience for the user in the form of a collection or hierarchy of windows. A GUI also implies the use of some pointing device, such as a computer mouse.

- Web user interface (Web UI)

A Web UI is any web-based system that provides a visual experience for the user in the form of a collection of interactive web pages. Usually, a Web UI is created from HTML pages, which can include HTML forms. Other UI technologies exist on web browsers, such as Java applet technology.

- Machine or device interface (for example, Web services)

A Web service is traditionally a portable web-based communication between systems. The systems can interact with a Web service by using SOAP messages that usually contain XML packets. These messages can be transported by using the HTTP or HTTPS protocols, which make these messages easier to transport across firewalls.

There are many different technologies for providing user input and feedback. Other types of Boundary Interfaces include:

- Touchpads
- Direct manipulation
- Joystick
- Interactive voice recognition
- Keypads
- Command line

The development of Boundary Interfaces is beyond the scope of this course. However, this module discusses some of the key architectural issues for GUI and Web UI systems.

There are four fundamental types of application components:

- Controller

These components provide a response to user actions, such as typing in a text field, selecting a menu item, or clicking on a button.

- View

These components provide an observable representation of the state of the system. Views are usually visuals, but they can also be auditory-based or character-based.

- Service

These components represent business services, such as use case workflow management.

- Entity

These components represent business entities (domain objects).

Figure 12-2 illustrates a generic diagram of the four fundamental types of components that constitute the functional elements of an application. As mentioned previously, controller and view components are UI components. Service and entity components map to the generic Design components of Robustness analysis.

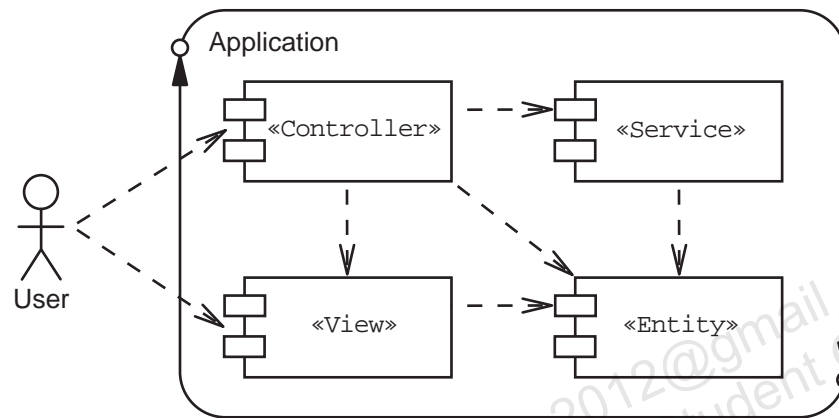


Figure 12-2 Generic Application Components

This module introduces you to the technologies that create controller and view components for GUI and Web UI systems.

Overview of the GUI in the Client Tier of the Architecture Model

This section explores the concepts and Java technologies for developing GUI applications. You will also see how to populate the Architecture model with the design choices for the Client tier.

GUI Screen Design

A GUI screen usually consists of a set of labels and data entry components, such as text fields. Figure 12-3 shows an example screen.

Customer Management Screen

First name:

Last name:

Phone:

Address

Street1:

Street2:

City:

State: ▼

Zip:

Figure 12-3 Customer Management Screen

The visual design of a GUI screen is a real art. It is beyond the scope of this course to discuss design and layout strategies for GUI screens. However, this module does discuss the Java technologies that support the creation of GUI screens (the View) and the event listeners that accept user actions (the Controller).

Java technology GUIs are usually constructed with the Swing framework. This framework provides a robust set of low-level GUI components, such as text fields, labels, buttons, list and tree components, tables, and so on.

A GUI screen is constructed from an organized collection of low-level GUI components grouped into panels, and panels can be grouped into higher-level panels. Finally, the panels are grouped into some sort of screen components such as a frame. Figure 12-4 illustrates this GUI component hierarchy.

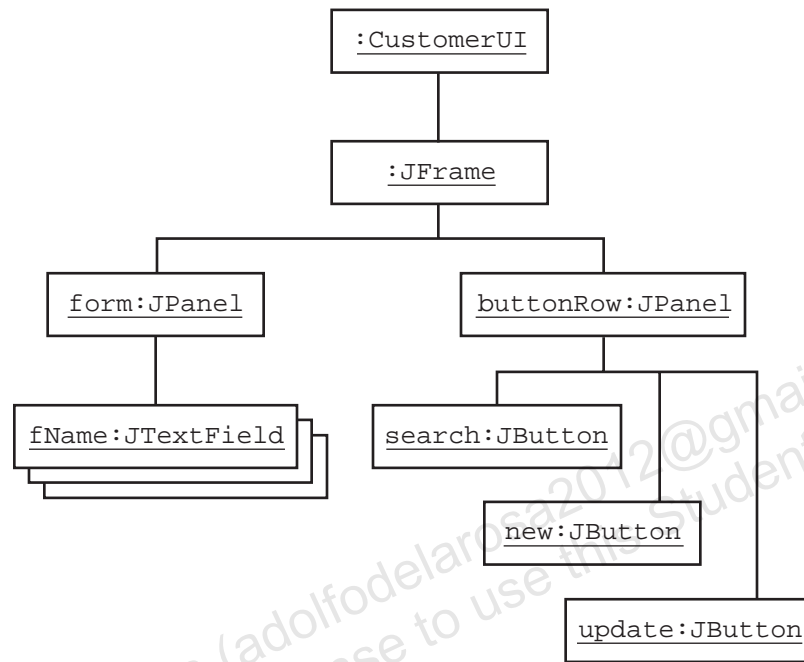


Figure 12-4 CustomerUI GUI Component Hierarchy

GUI Event Model

A GUI must also provide mechanisms to receive user actions, such as clicking on a button, entering text in a text field, tracking the mouse over a graphic, and so on. In the Swing framework, this function is achieved using a component called a listener. A listener responds to the type of operations allowed for a given GUI component. For example, a button can be *pressed* by clicking a button on your mouse. Swing includes about a dozen different listeners. It is beyond the scope of this course to discuss the subject of listeners.

When you create an implementation of a listener, you register that listener with the appropriate GUI component. Whenever an action occurs on that GUI component (such as pressing a button), your listener object is triggered.

Figure 12-5 illustrates the activation of an ActionListener object when the Update button is pressed. The Swing framework provides for the operating system hooks that intercept the mouse button action. When a button is pressed the GUI component for that button is notified, which triggers any and all ActionListener objects that have registered with that button. For example, the Update button listener might store the name, phone number, and address data into the Customer object and then call the save method on the CustomerSvc class.

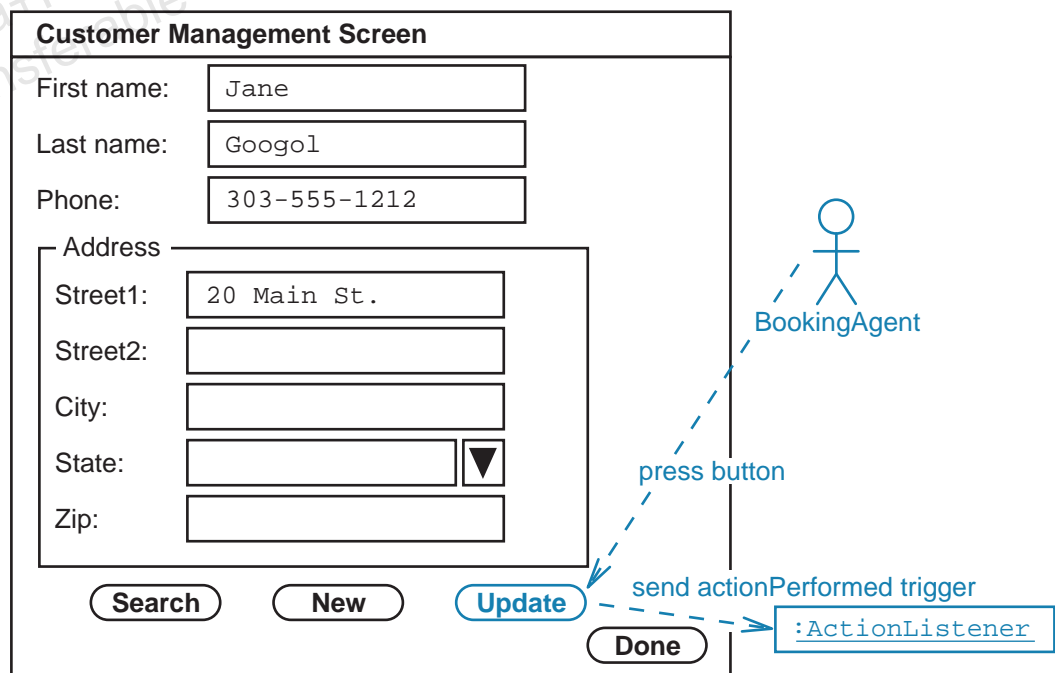


Figure 12-5 Java Technology Event Model

The listener objects that you create are implemented using inner classes in your GUI Presentation component class. This design allows the listener objects direct access to the private data in the Presentation component. For example, pressing a button might change the values of a list within the Presentation component. The listener objects are directly associated with the low-level GUI component for which they are registered. Figure 12-6 illustrates this collaboration between the top-level Presentation component, CustomerUI, the low-level GUI components, and the listener objects.

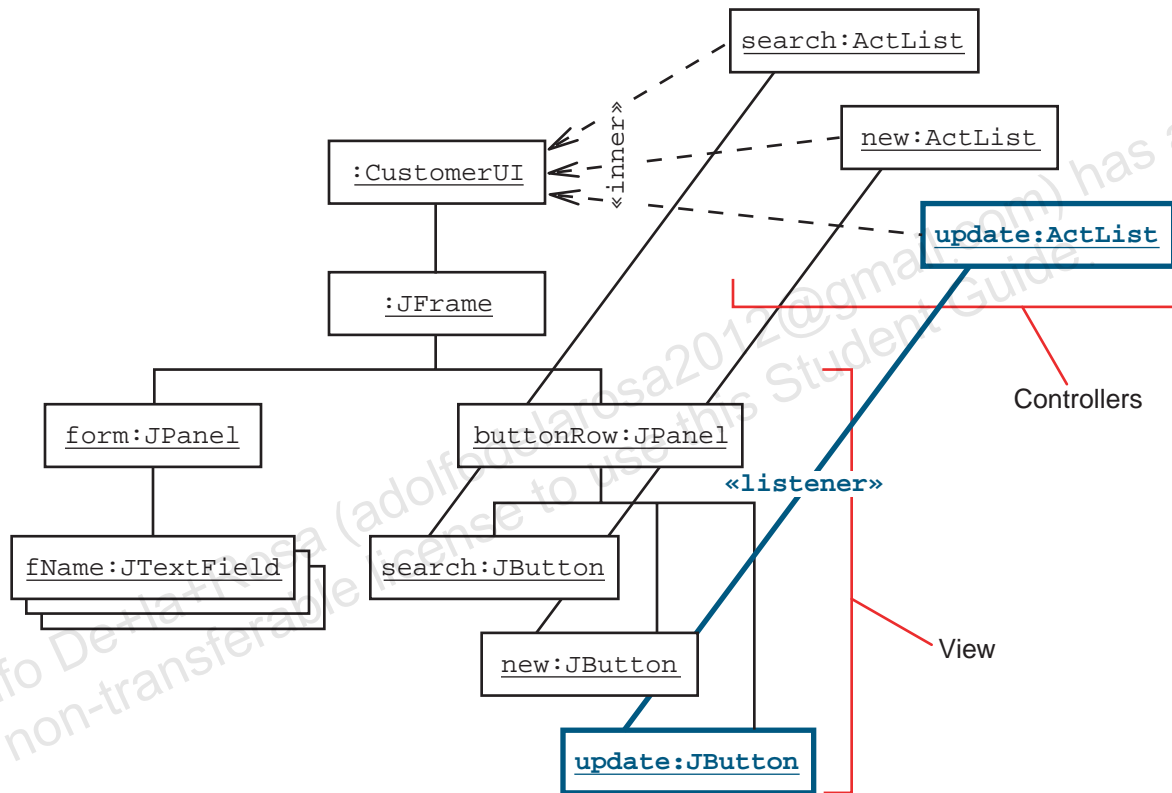


Figure 12-6 GUI Listeners as Controller Elements

The low-level GUI objects comprise the View elements of the PAC agent and the listener objects comprise the Controller elements of the agent.

MVC Pattern

MVC is another GUI architectural pattern. This pattern is an example of the Separation of Concerns principle in which the Model elements (the business Entities and Services) are kept separate from the GUI Views and Controller mechanisms. The MVC pattern uses a notification mechanism similar to that used in Swing. A View component can register with the Model to listen to events when the Model changes. Figure 12-7 illustrates the responsibilities of each element in the MVC pattern.

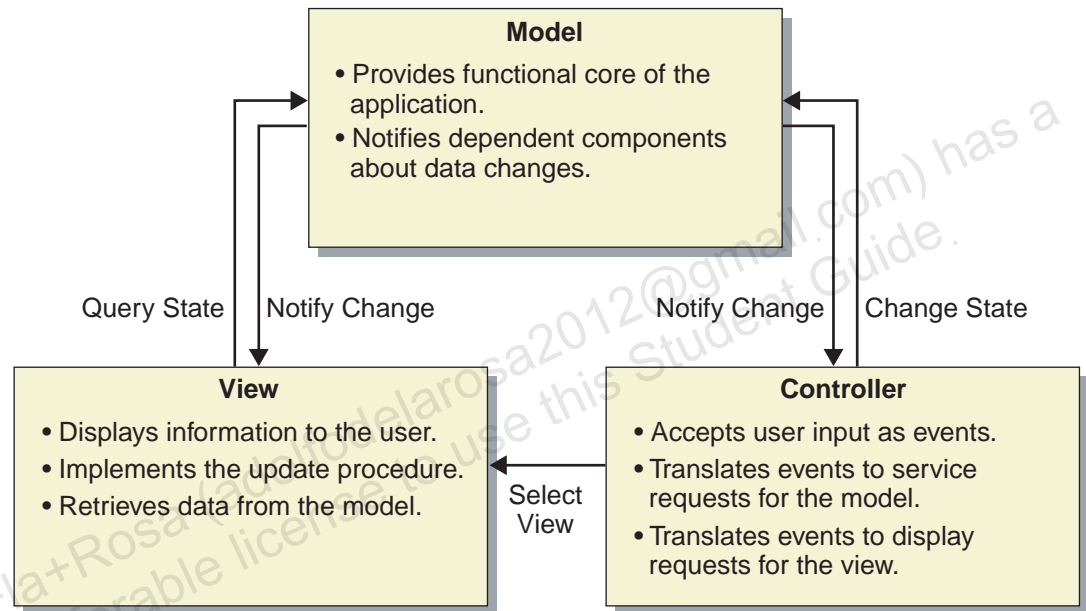


Figure 12-7 MVC Pattern

The main difference between PAC and MVC is that PAC usually organizes the top-level structure of all GUI agents in the application; whereas, MVC usually provides a separation between the low-level views of the system and the system data. PAC and MVC can coexist within the same GUI.

Figure 12-8 provides another view of the MVC pattern. This shows how each of the MVC components maps to the four fundamental components. The important thing to note is that the Model component encapsulates the behavior of both the Entity and Service aspects of the application.

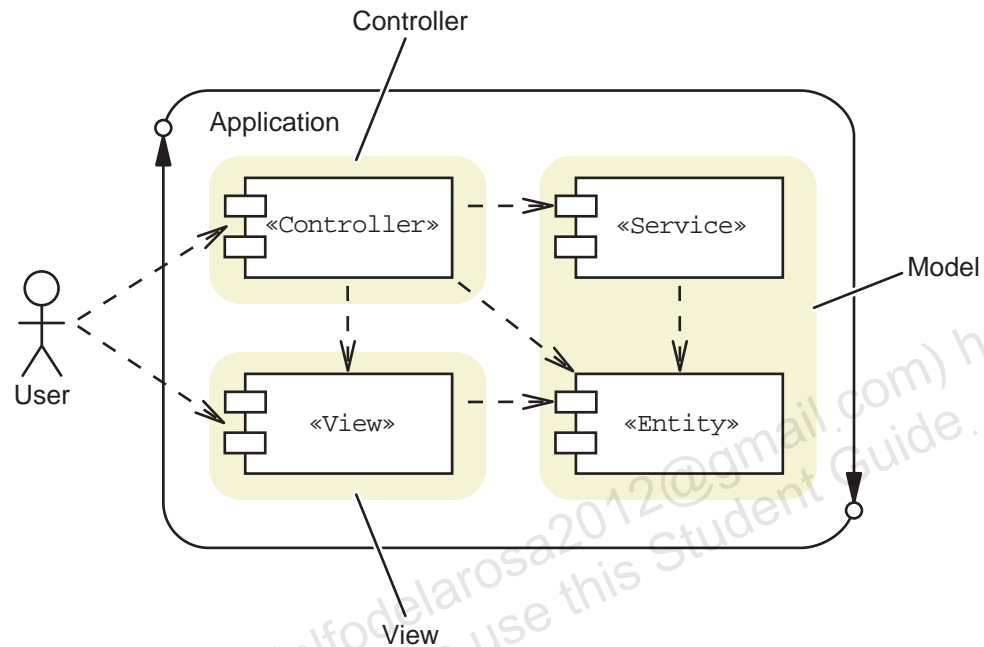


Figure 12-8 MVC Component Types

An important advantage of the MVC pattern is the clear separation of the data from the visual representation of that data. Figure 12-9 provides an example of tabular data and three views of that data.

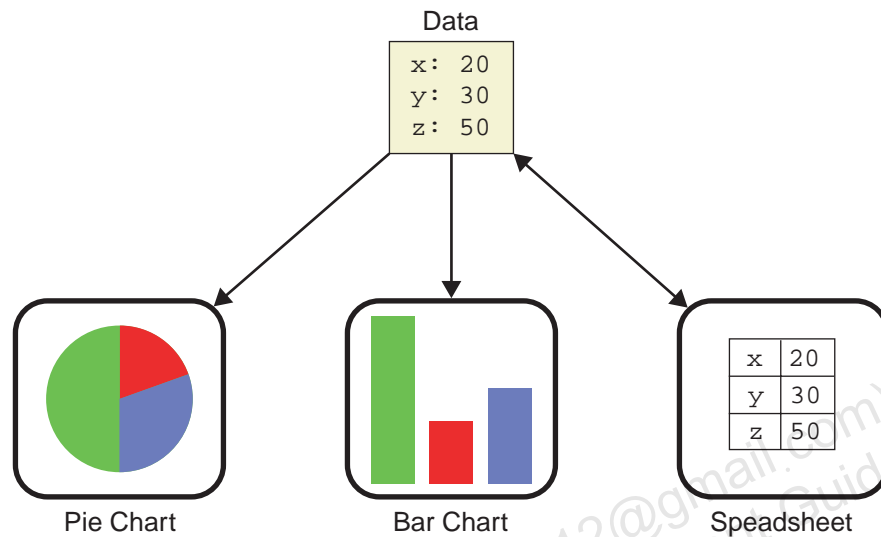


Figure 12-9 Example Use of the MVC Pattern

Overview of the Tiers and Layers Package Diagram

Figure 12-10 illustrates an example tiers and layers package diagram for the HotelApp application. For the HotelApp, the Swing GUI components will be used to implement the Client tier components. Swing provides the framework on which the Application layer components are created; this is the Virtual Platform (VP) layer. The Upper Platform (UP) layer specifies the container for the VP components; in this example, J2SE (v1.4) provides the container for Swing GUI components. The Lower Platform (LP) and Hardware Platform (HP) layers are not so significant for the Client tier because J2SE can execute on almost every OS and hardware platform.

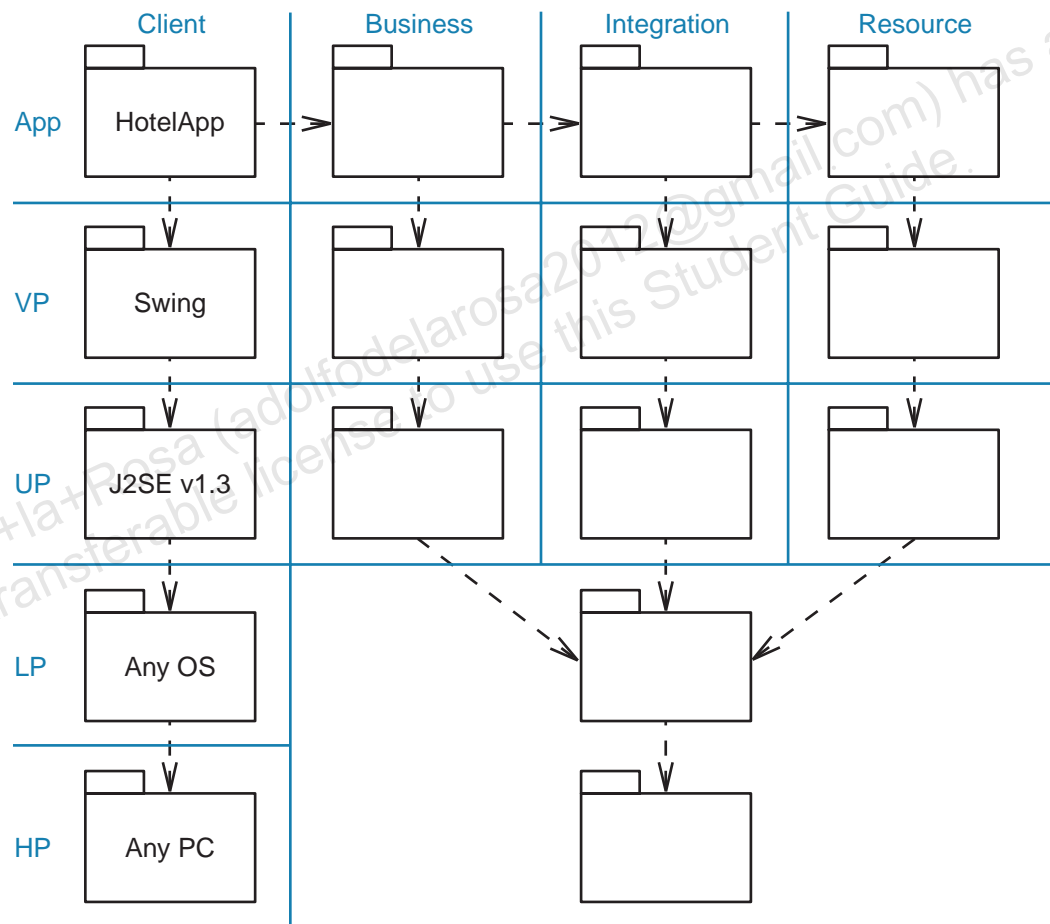


Figure 12-10 A Partial Tiers and Layers Package Diagram for the HotelApp

Note – In this figure, the Business, Integration, and Resource tiers have not been completed. The technologies for these tiers are described later in the module.



Overview of the Web UI in the Presentation Tier of the Architecture Model

This section explores the concepts and Java technologies for developing Web UI applications. You will also see how to populate the Architecture model with the design choices for the Presentation tier.

A Web UI provides a browser-based user interface. In a Web UI, the Client tier is comprised of the web browser. The browser provides the View to the user by rendering an HTML page. That page might include links or HTML forms that enable the user to perform actions within the application; and these actions take the form of an HTTP request to the web server. The components that intercept HTTP requests exist on the Presentation tier. The Presentation tier is sometimes called the Web tier for this reason.

Web UIs have the following characteristics:

- Perform a few large user actions (HTTP requests).

A Web UI is controlled by a few, large-scale (coarse-grained) user actions. This action takes the form of an HTTP request which is sent to the web server for processing. Such actions usually require data from the user; HTML forms provide the mechanism to collect such data and send it to the HTTP request.

Small-scale user actions can be handled using JavaScript™, Flash, or Java applet technologies. These technologies are not discussed in this course.

- A single use case is usually broken into multiple screens.

Typically, a Web UI is constructed from a sequence of web pages that include HTML forms and dynamic views of the business data.

- There is often a single path through the screens.

There is usually a dominate flow through the Web UI screens that guides the user. There will be alternate paths (such as login screens) for alternate flows through a use case.

- Only one screen is usually open at a time.

Unlike a GUI, a web browser can only show one screen at a time. Because of this, a Web UI usually is comprised of multiple screens. It is possible to create a web page with multiple paths of functionality, but this might confuse the user.

Web UI Screen Design

The following considerations affect the design of Web UI screens:

- A Web UI tends to be constructed as a sequence of related screens.
The Web UI page sequence usually implements the workflow of a single use case.
- Each screen is a hierarchy of UI components.
Each Web UI page can include much information. Typically, a Web UI page will include an HTML form that tells the Web browser to create corresponding GUI components, such as text fields.
- A Web UI screen presents the user's view of the domain model as well as presents the user's action controls.
A Web UI page presents the state of the application using HTML features, such as tables. A Web UI page can use HTML forms to permit users to interact with the application.
- It is rare that a screen can be reused by multiple use cases.
Because Web UI screens tend to be designed specifically for a single use case, this usually prevents these components from being reused in other use case workflows. There are a few notable exceptions. For example, login or *create user* screens might be reused in a variety of use cases.

Figure 12-11 shows an example sequence of Web UI pages for the Hotel Reservation System. This sequence implements the Create a Reservation Online (use case number E5) use case.

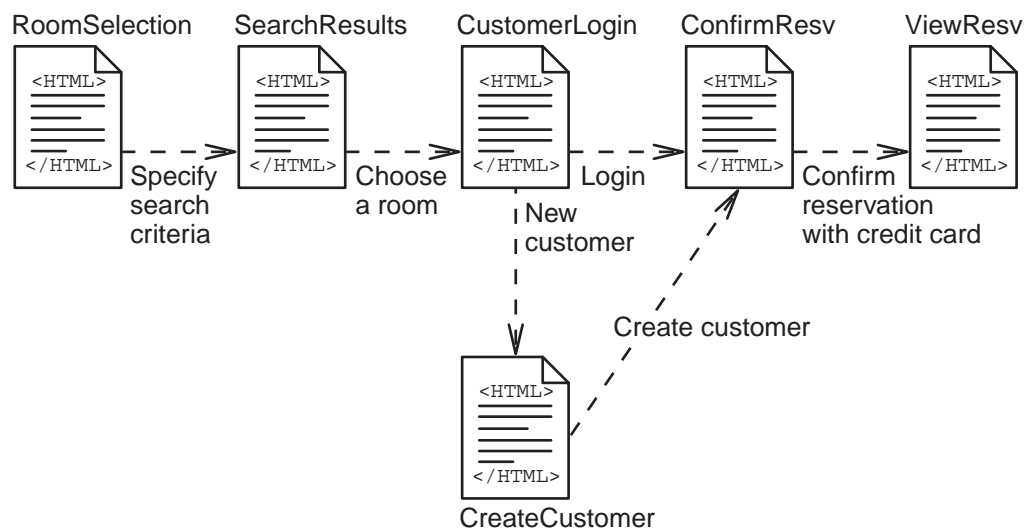


Figure 12-11 Example Web Page Flow

Web UI pages are based on HTML forms. HTML includes tags to specify many useful UI components, such as text fields, drop-down lists, radio buttons, and so on. Code 12-1 shows an example HTML form. A detailed discussion of HTML forms is beyond the scope of this course.

Code 12-1 Partial Web UI Form Example

```
1  <FORM ACTION='makeResv' METHOD='POST'>
2    <INPUT TYPE='hidden' NAME='action' VALUE='roomSearch'>
3    Enter arrival date: <INPUT TYPE='text' NAME='arrivalDate'>
4    <BR>
5    Enter departure date: <INPUT TYPE='text' NAME='departureDate'>
6    <BR>
7    Select room type:
8    <SELECT NAME='roomType'>
9      <OPTION VALUE='Single'> Single
10     <OPTION VALUE='Double'> Double
11     <OPTION VALUE='Suite'> Suite
12   </SELECT>
13   <BR>
14   <INPUT TYPE='submit' VALUE='Search...'>
15 </FORM>
```

Web UI Event Model

A Web UI application has a radically different event processing model than a GUI application. There are two types of events:

- Micro events can be handled by JavaScript™ technology code.
Small-scale events occur as the user clicks on UI components, enters text, and changes the focus from one UI component to another. These events can only be handled by the web browser. JavaScript™ technology is useful for handling such micro events.
- Macro events are handled as HTTP requests from the web browser to the Web server.

Large-scale events occur as the user clicks on a link or submits an HTML form. These events are sent to the web server as an HTTP request. This request will include all of the HTML form data, if any.

Figure 12-12 illustrates a macro event in the WebPresenceApp. In this example, the customer uses the RoomSelection page to enter the data that performs a search for a hotel room during a certain time frame. When this form is submitted, an HTTP request is generated from the browser to the Web server. There are components within the WebPresenceApp that respond to this action and generate another view of all of the possible rooms that satisfy the search criteria.

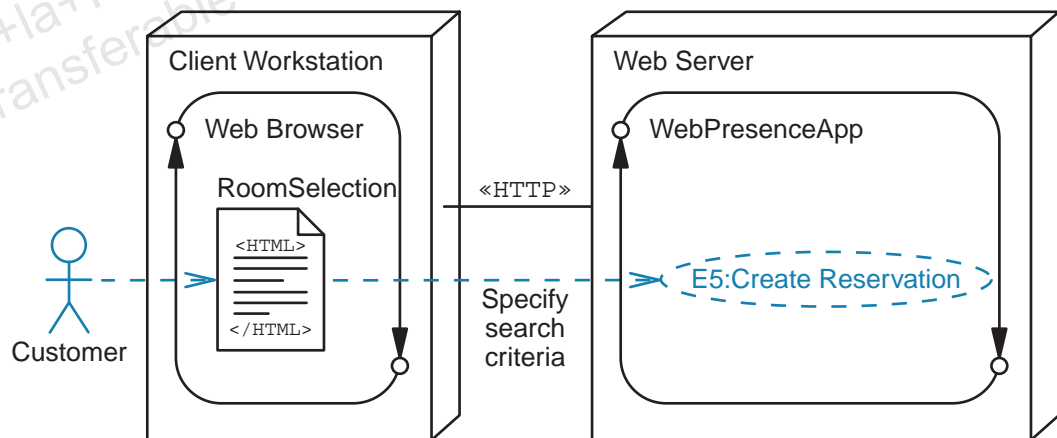


Figure 12-12 Web UI Event Model

This section shows how to architect web application using Java technologies such as servlets and JavaServer Pages technology, but it does not show how to develop these web components.

The WebMVC Pattern

A variation on the GUI MVC pattern is usually used to architect the Web UI components, and this pattern can be called Web Model-View-Controller (WebMVC). Similar to the GUI version of this pattern, WebMVC provides the Separation of Concerns between the Model, View, and Controller elements of the web application. Figure 12-13 describes the responsibilities of these elements.

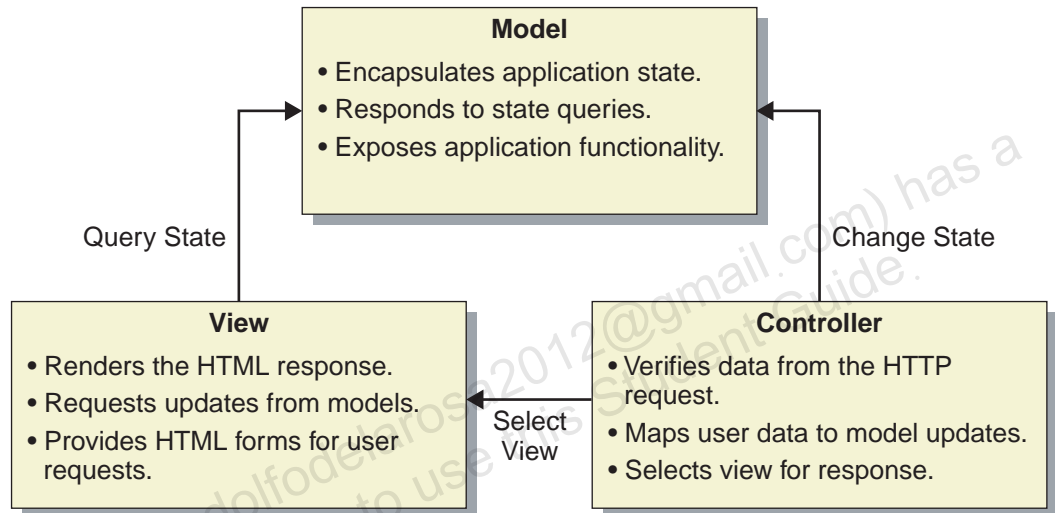


Figure 12-13 WebMVC Pattern

The main difference between the GUI MVC and the WebMVC is that there is no mechanism to update the Views when the Model changes¹. Such functionality would have no meaning for a web application because changed data in the Model cannot be pushed to the Web browser which renders the View. However, even with that limitation, the WebMVC pattern is powerful because of the results of separating these three types of components. For example, it is not uncommon for a Web site to change the look and feel of the site several times a year. By separating the View elements from the Controller elements, the development team can focus on the look-and-feel changes to just the View elements.

1. This statement is not quite true. It is possible to build web pages that periodically request updates from the server. This is called *client pull*. There is also a technique called *server push*, but this requires technology beyond the capabilities of the Web browser and the HTTP protocol. It usually requires applet to server communication.

Figure 12-14 provides another view of the WebMVC pattern. This shows how each of the WebMVC components maps to the four fundamental components. The important thing to note is that View and Controller components have been specified as JSP technology pages and servlets, respectively.

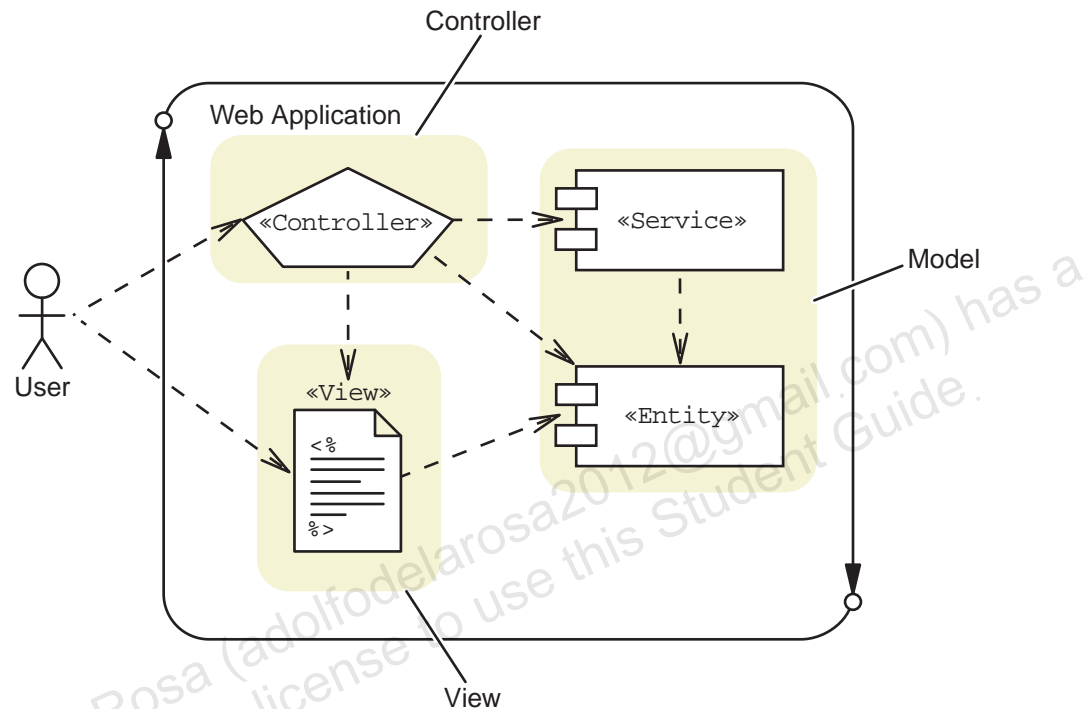


Figure 12-14 WebMVC Pattern Component Types

The structure of the Web application in Figure 12-14 is usually referred to as the *Model 2 architecture*. The Model 2 architecture specifies the responsibilities of each Web component type:

- Java servlets act as a Controller to process HTTP requests:
User actions in a web application are coarse-grained. A user action is a complete HTTP request, which is usually in response to the user submitting an HTML form. This request contains all of the data on that form. The servlet Controller must perform the following operations:
 - Verify the HTML form data
 - Update the business Model
 - Select and dispatch to the next View (the HTTP response)

There is usually only one servlet Controller for every use case in a web application. The servlet is responsible for managing a user's flow through the Web UI screens for a given use case.

- JavaServer Pages technology acts as the Views that are sent to the user.

JSP technology pages generate HTML text, but have access to the data in the business model. This enables dynamic web page creation. For example, the SelectRoom JSP technology page might use a Property object to retrieve the list of rooms which are then displayed in a list box in the SelectRoom HTML form.

- Java technology classes (whether local or distributed) act as the Model for the business services and entities.

The Model elements are the business logic and entity components. These components can solely reside on the web container, in a web-centric architecture, or as remote objects in an enterprise architecture. The architecture of the Model elements is the focus of the next module.

The Client tier components are HTML pages that are rendered by a web browser. The JSP technology page components generate these HTML pages in the Presentation tier.

Overview of the Tiers and Layers Package Diagram

Figure 12-15 illustrates the Client and Presentation tier components for the WebPresenceApp.

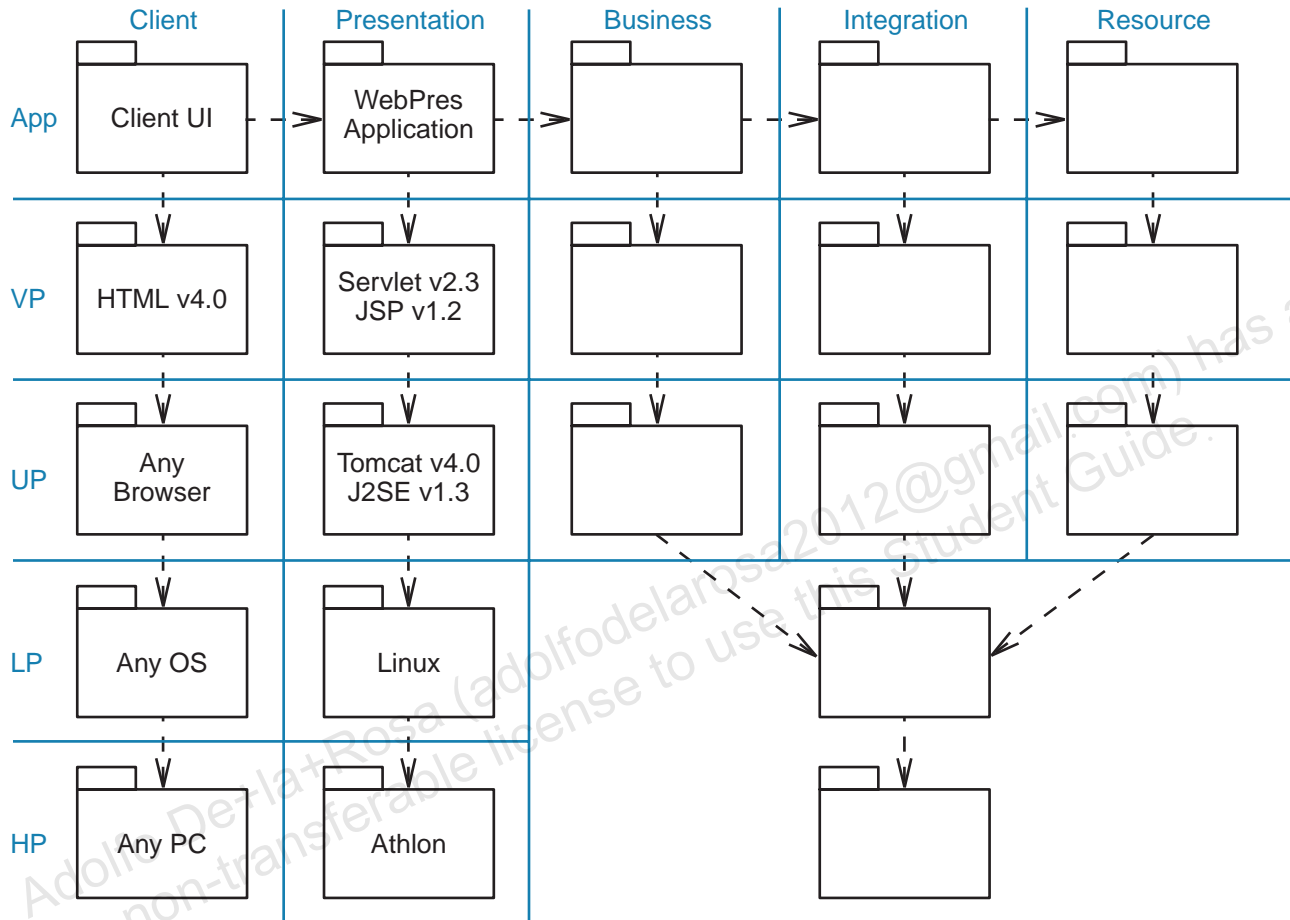


Figure 12-15 A Partial Tiers/Layers Package Diagram

Introducing the Business Tier

The Business tier primarily contains:

- Entity components
- Service components
- Perform validation of business rules
- Perform updates on the entity components

This tier contains the business entity components derived from your domain class model and the business services components discovered using Interaction diagrams.

The Business tier may be accessed by:

- Local components
- Remote components, for example:
- Remote Method Invocation (RMI)
- Web service protocols

This tier should not need to be changed in order to provide access to a wide range of boundary components.

Exploring Distributed Object-Oriented Computing

This section discusses how to architect the access to remote services. In the past two decades these technologies were created:

- CORBA

Common Object Request Broker Architecture (CORBA) is a specification from the Object Management Group. This is a rich and complex specification enabling the interaction of applications written in different languages. There are many vendor implementations of the CORBA specification. CORBA uses a communication protocol called Internet Inter-ORB Protocol (IIOP).

- RMI

Remote Method Invocation (RMI) was created by Sun Microsystems, Inc., for use in Java technology applications. RMI is a rich, yet simple, remote object environment. RMI enables the interaction of applications written using Java technology. RMI uses a proprietary network protocol, called Java Remote Method Protocol (JRMP).

Java technology also provides a mechanism to interface with CORBA objects.

- EJB technology

EJB technology is a specification from Sun Microsystems, Inc., for use in large-scale Java technology applications. EJB technology is a rich and complex specification. There are many vendor implementation of the EJB specification. EJB technology enables the interaction of applications written for Java technology, but it also has a CORBA bridge. EJB technology uses a network protocol that works with RMI and CORBA, called RMI-IIOP.

- Web services (SOAP)

Simple Object Access Protocol (SOAP) is a specification from the World Wide Web Consortium (W3C). SOAP is more of a remote procedure mechanism than it is a remote object mechanism.

Note – This is not a programming course. You will not see how to write the code to perform remote service calls, but you will see how to architect RMI applications.



Local Access to a Service Component

Client application components need services provided by some object, which is the service provider. If the service is local to the client application, then the communication between the client and the service provider is through a local method call. Figure 12-16 illustrates this.

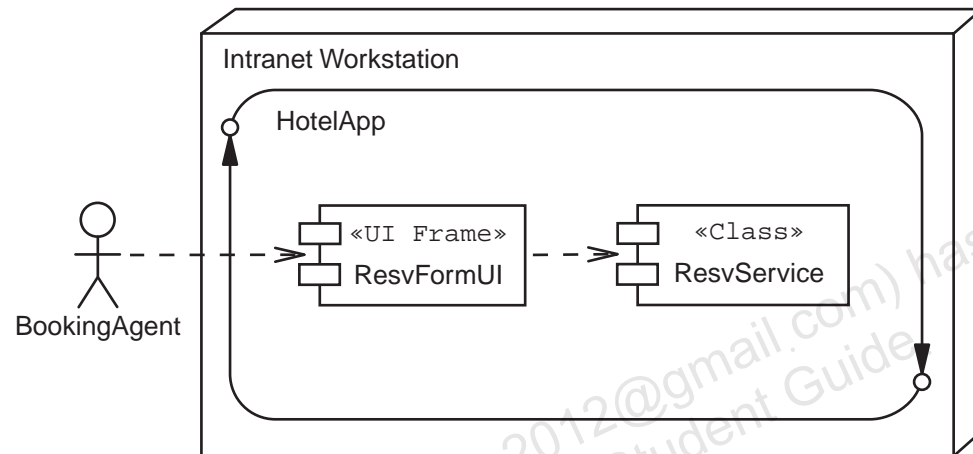


Figure 12-16 Local Access to a Service Component

In this architecture the client code is tied directly to the class that implements the service. This is a very rigid software structure.

Alternatively, you could have coded the client component to an interface for the service, which can be implemented by some class. This architectural principle is called Dependency Inversion Principle, because the client component depends on an interface rather than an implementation. Figure 12-17 illustrates this.

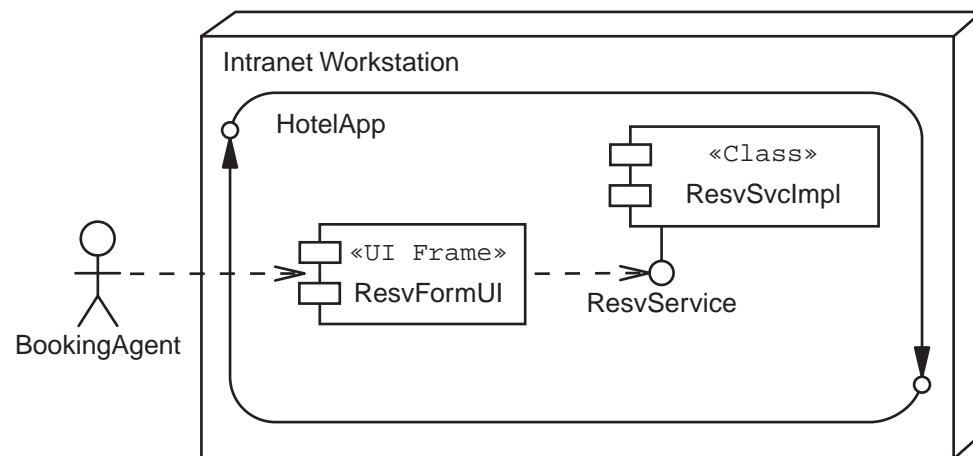


Figure 12-17 Applying the Dependency Inversion Principle

Figure 12-17 uses a line with a small circle to indicate the interface that the service component implements. In Java technology, this single component would be coded with two elements: the `ResvService` interface and the `ResvServiceImpl` implementation class. Figure 12-18 shows these elements.

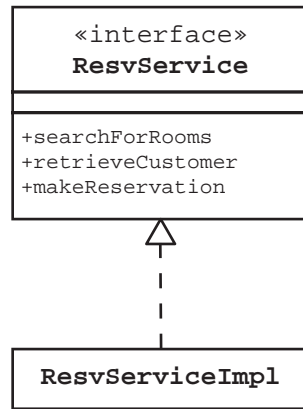


Figure 12-18 Class Diagram of the `ResvService` Interface and Implementation

Remote Access to a Service Component

To make this service remote, you would move the implementation element to a remote application server. Figure 12-19 illustrates an abstract version of this architecture.

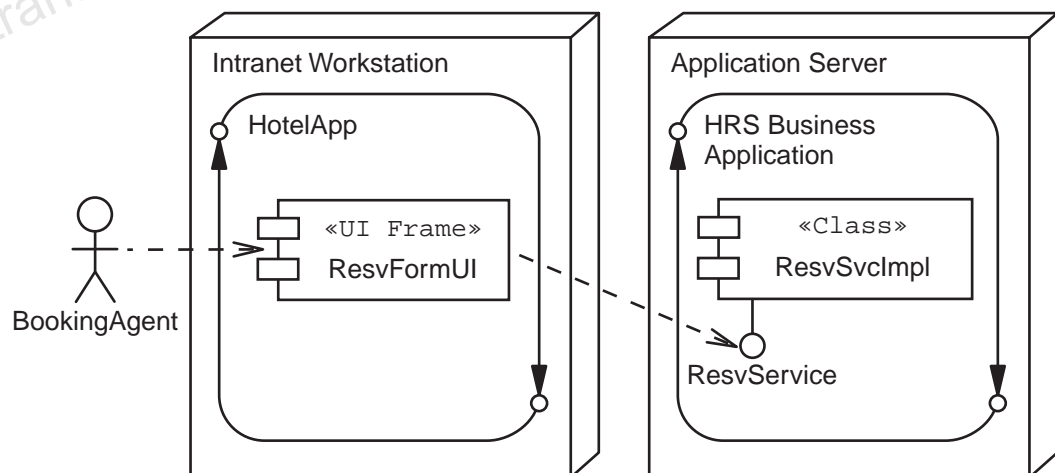


Figure 12-19 An Abstract Version of Accessing a Remote Service

Remote Access Using RMI

The big question to ask about Figure 12-19 is this: How does the client component communicate with the remote service? This requires additional infrastructure as well as a network communication protocol. In RMI, you need two additional components: a stub and a skeleton. The client component communicates with stub component. The stub implements the service interface, so that the client thinks that it is talking to the real service. In fact, the stub communicates to the application server through the skeleton component using the JRMP network protocol. It is the skeleton component that talks to the actual service implementation.

Since the J2SE v1.2 platform, the RMI mechanism no longer requires the skeleton component. Figure 12-20 illustrates this simplified architecture. Notice that the diagram shows the stub communicating directly with the business application component; the diagram does not show the real component that the stubs talk to directly because this is hidden from the system as part of the RMI runtime environment.

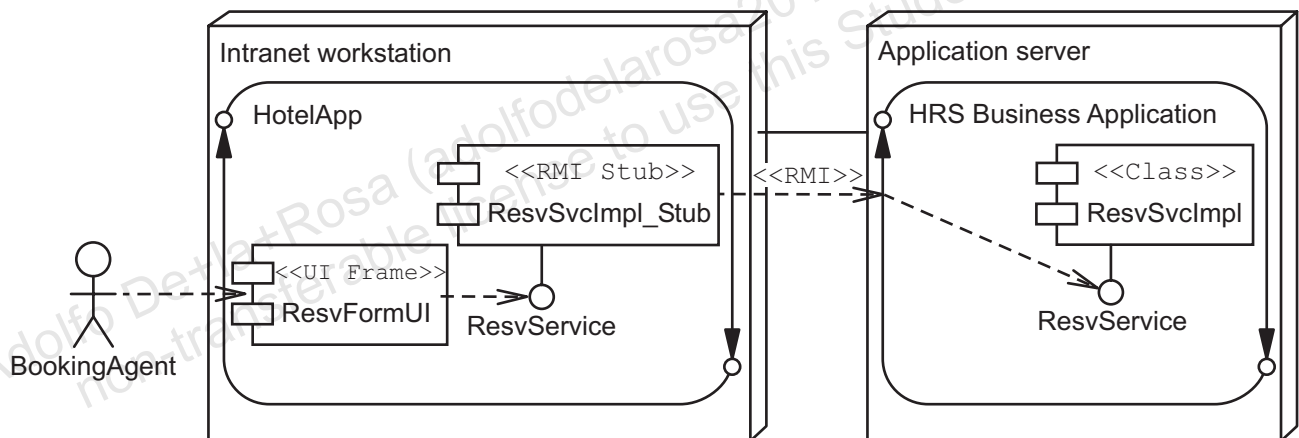


Figure 12-20 Accessing a Remote Service Without a Skeleton Component

Parameter Passing Using RMI

Methods on a service might pass primitive data values, such as integers, floating point numbers, or characters. They can also pass objects. To send an object across a network requires that the data in the object be sent byte-by-byte; this process is called object serialization. Figure 12-21 illustrates this.

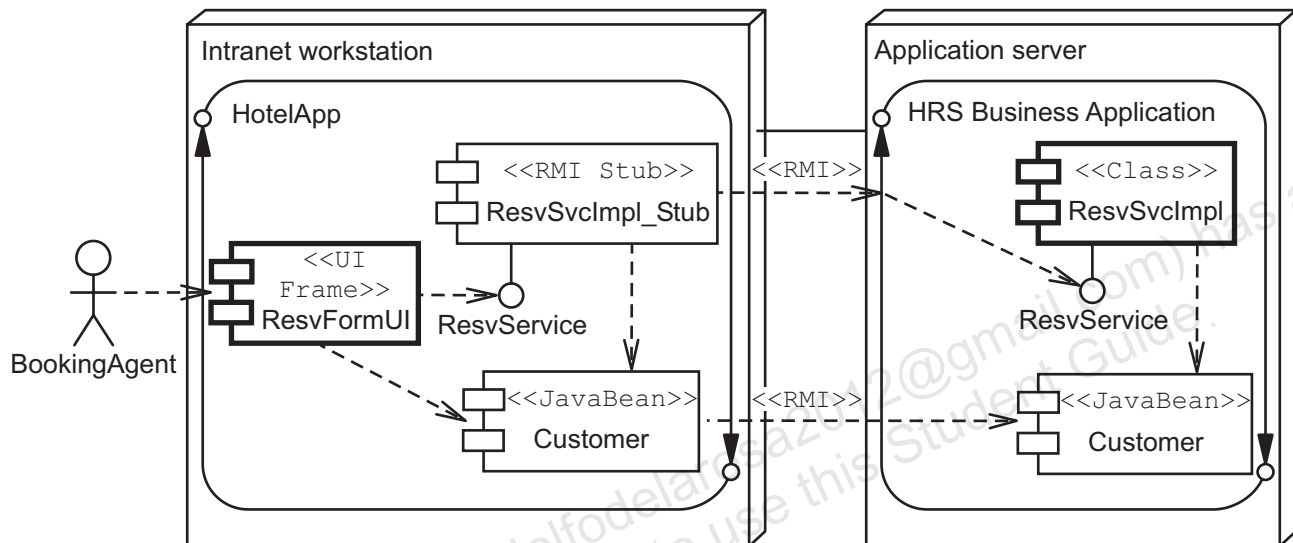


Figure 12-21 RMI Uses Serialization to Pass Parameters

It is important to realize that if an object is created on the application server, when that object is sent to the client application, a new object is created. These two objects are duplicates at the moment of transmission, but thereafter the two objects can be changed independently.

Service Lookup Using RMI

Finally, the client component must find the remote stub by using another remote application called the RMI registry. Figure 12-22 illustrates this.

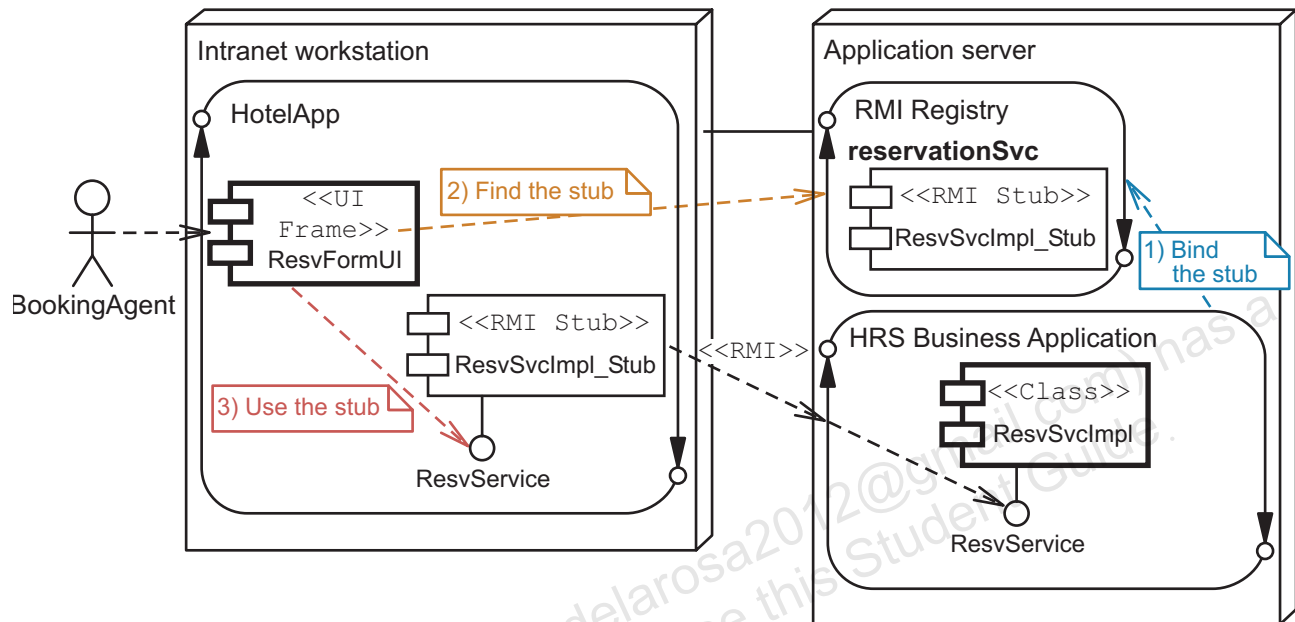


Figure 12-22 RMI Registry Stores Stubs for Remote Lookup



Note – The previous four figures show an application on the Application server called “HRS Business Application.” This application manages the creation of the remote business service objects. This creation process is often called *launching* the remote services. This launch program is usually quite simple; it instantiates each service object implementation and binds that service to the RMI registry using the service name.

Overview of the Detailed Deployment Diagram

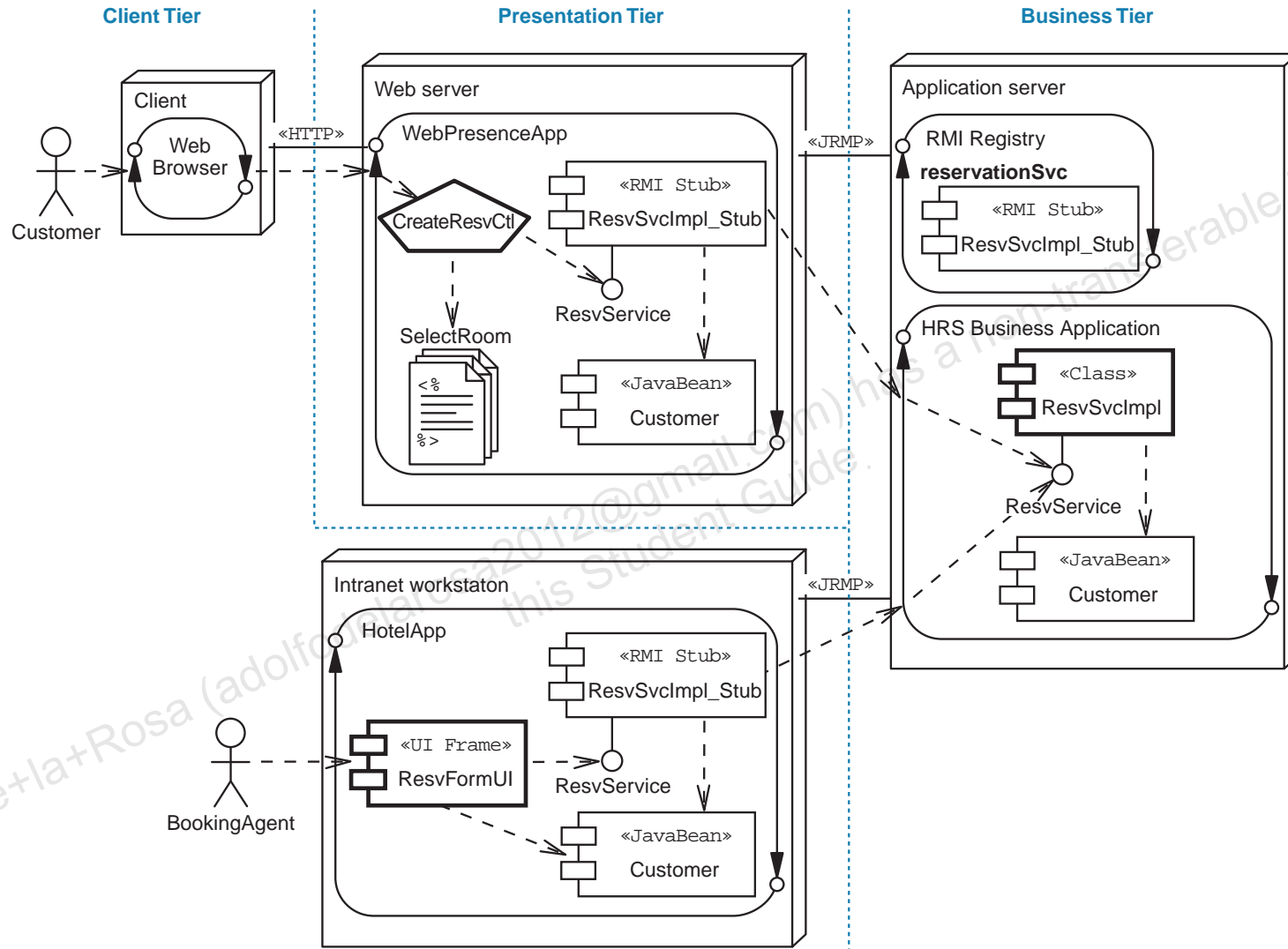


Figure 12-23 Example Detailed Deployment Diagram

Overview of the Tiers and Layers Package Diagram

Figure 12-24 shows an example tiers and layers Package diagram for the HotelApp application. The HotelApp is a fat client that communicates directly with the application server; therefore, it does not require a Presentation tier. The HotelApp client tier uses Java technologies, such as Swing, to build the user interface.

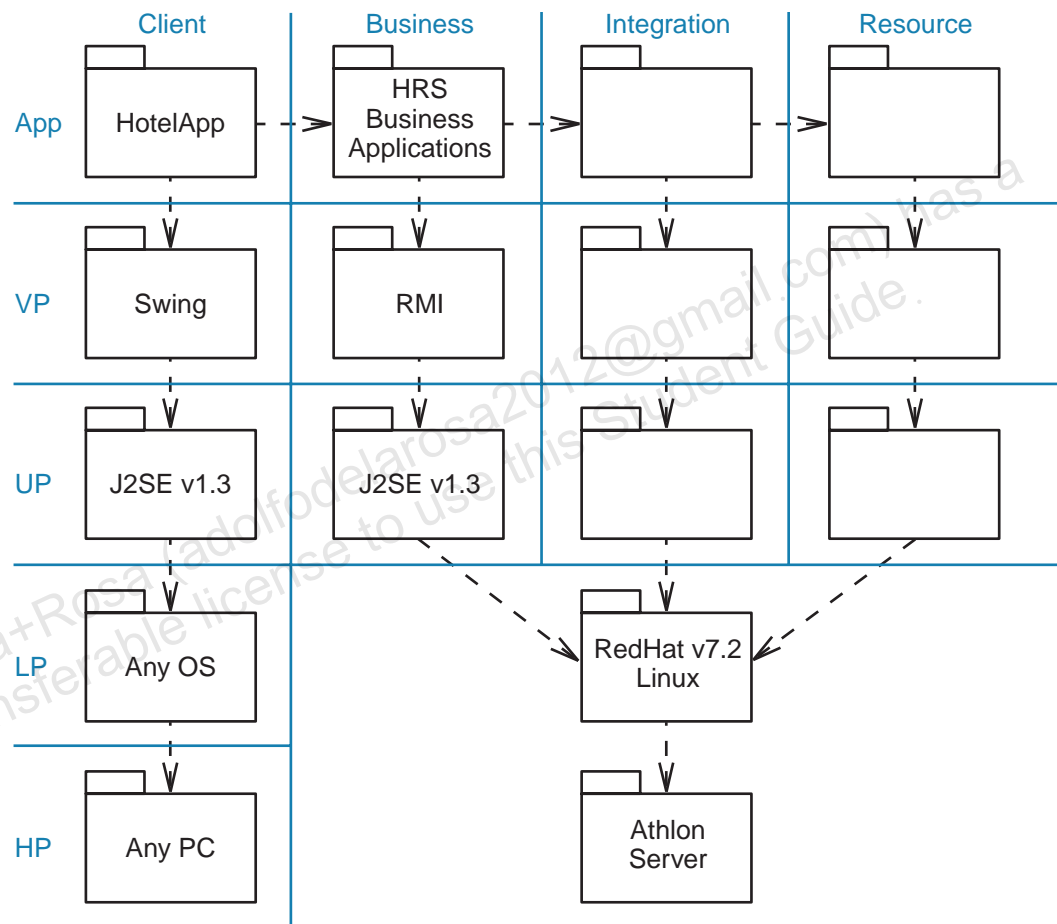


Figure 12-24 Tiers and Layers Diagram for the HotelApp

Note – In this figure (and the next), the Integration and Resource tiers have not been completed. The technologies for these two tiers are described later in the module.



Figure 12-25 shows the tiers and layers Package diagram for the WebPresenceApp.

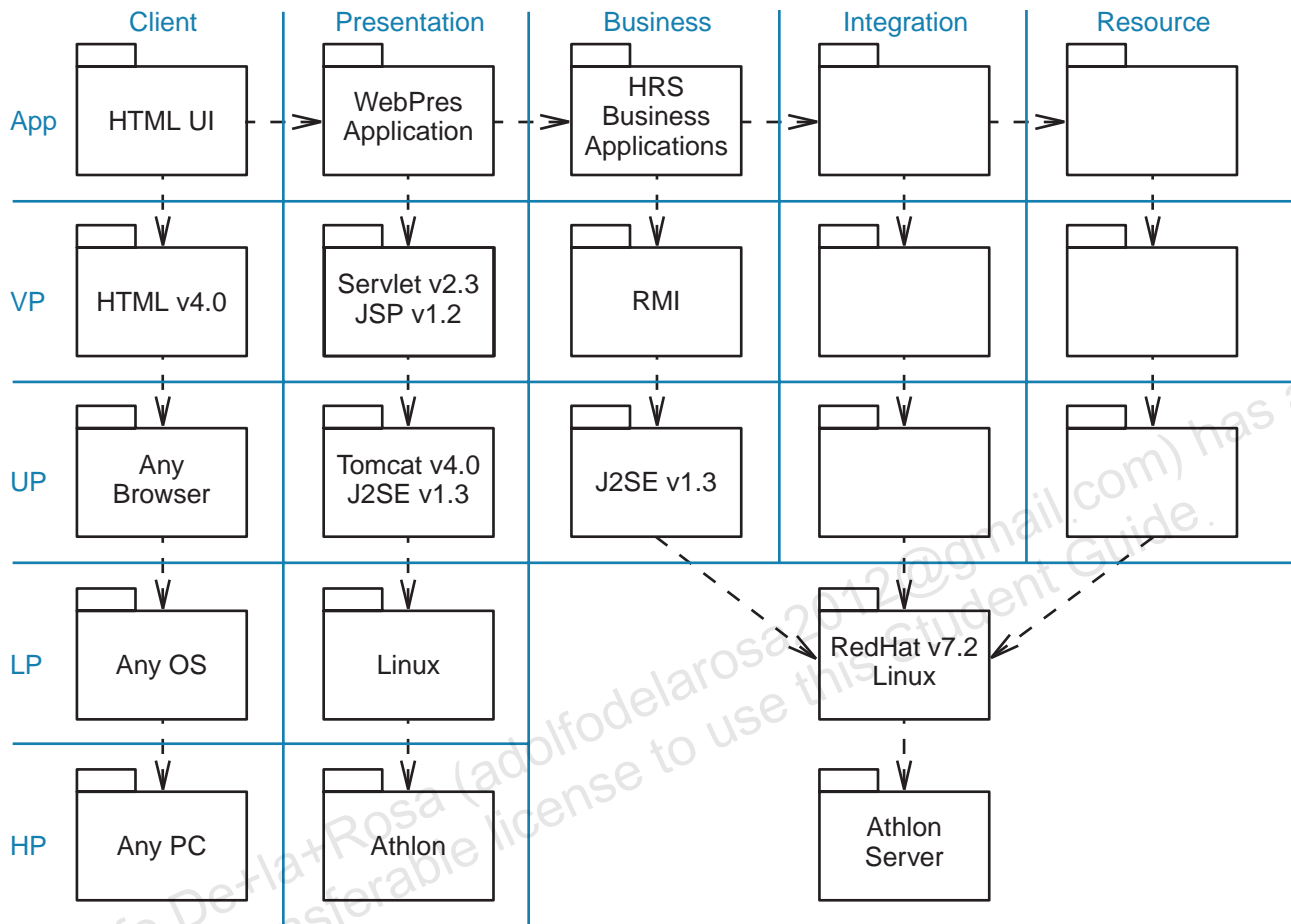


Figure 12-25 Tiers and Layers Diagram for the WebPresenceApp

Introducing the Resource and Integration Tiers

Exploring the Resource Tier

This section explores the concepts and RDBMS technologies for developing a persistence mechanism. You will also see how to populate the Architecture model with the design choices for the Resource tier.

Most applications require storing information from one execution of the application to another and the ability to share that information with multiple, concurrent users. This section discusses the concepts and issues that solve these needs.

The Resource tier includes:

- Database
- File
- Web service
- Enterprise Information System (EIS)

Exploring Object Persistence

Persistence is “The property of an object by which its existence transcends time and space.” (Booch OOA&D with Apps 517)

Booch’s definition of persistence includes two important points. A *persistence object* is:

- An object that exists beyond the time span of a single execution of the application

When an application starts up, objects are created. If these objects are transient, the objects no longer exist and all of your data is lost when the application shuts down. If you restart the application, the objects from the previous execution are gone.

Alternatively, the system can use persistent objects. When an application creates a new persistent object, the system saves that object. When the application is restarted, you can retrieve that same object.

- An object that is stored independently of the address space of the application

Transient objects only exist within an execution of the application. Therefore they reside solely in the address space of the executing applications.

Persistent objects exist outside of the address space of the executing application. The application must have a mechanism to load a persistent object into its address space. Similarly, new persistent objects must be stored in some external storage before the application shuts down or the object will be lost.

Persistence Issues

Here are a few of the persistence issues that must be addressed:

- Type of data storage

How is the data stored? There are many possibilities including: flat-file, eXtensible Markup Language (XML) file, relational DBMS, object-oriented DBMS, and so on.

Some systems might start with one type of storage, such as a flat-file, and later migrate to a more robust storage mechanism, such as a relational database. Your application should be written to shield the business code from these changes (to avoid having to rewrite the business code each time the storage type is changed). In this module, you are introduced to the *Data Access Object* pattern which you can use to hide the type of data store from the application. This pattern is discussed in “Details of the DAO Pattern” on page 12-45.

The system in the Hotel Reservation System case study uses a RDBMS as its data store.

- Data schema that maps to the Domain model

The persistent objects in an application usually correspond to the entities specified in the Domain model. You must create a *data schema* that maps the domain entities, their relationships, and their attributes to the selected data store. SQL’s data definition language (DDL) specifies the data schema for a relational database. Similarly, a document type definition (DTD) of XML schema is used to specify the data schema for an XML file.

Straight-forward, but simple, object to table mapping guidelines is described in “Creating a Database Schema for the Domain Model” on page 12-35.

- Integration components

There are roughly two types of integration components.

There are technologies that provide hooks to communicate with the external data store. For example, Java DataBase Connectivity (JDBC) is the Java technology used to communicate between a Java technology program and a relational database. The details of using JDBC are not covered in this course.

There are technologies that provide the data access from the Business tier components. The DAO pattern is an example of this; see “Details of the DAO Pattern” on page 12-45.

- **CRUD operations: Create, Retrieve, Update, and Delete**

The CRUD operations are the fundamental operations on a data store. The create operation enables you to insert a new object into the data store. The retrieve operation enables you to find an existing object in the data store, using a unique identifier.² The update operation enables you to change the data attributes of the object in the data store. The delete operation enables you to remove the object from the data store.

Creating a Database Schema for the Domain Model

Defining the database schema is usually done in two phases: creation of a logic schema and the creation of a physical schema. This section describes a simple strategy to map OO entities to DB tables.

The logical schema is represented by an entity-relationship (ER) diagram. This section presents a strategy to map Domain entities to relational tables. The ER diagram represents the Domain entities as relational tables. Each instance of a Domain entity class is stored as a single row in the corresponding DB table. The attributes of the Domain entities are represented as fields within the tables. The associations between Domain entities are represented as foreign-key relationships between tables.

The physical ER diagram takes the logical ER diagram and adds data types on fields, indexes on tables, data integrity constraints, and so on. The physical ER diagram is not discussed in this course.

The logical ER diagram can be created during the Analysis or Architecture workflows. The physical ER diagram is usually evolved from the logical ER diagram during the Design workflow.

The strategy for converting the Domain model into a logical ER diagram is:

2. You can also select a set of objects using a query.

1. Convert each Domain entity into a table.
2. Specify the primary key for each table.
3. Create ER associations, either one-to-many or many-to-many.

Simplified HRS Domain Model

Figure 12-26 presents a simplified Domain model for the Hotel Reservation System. The following sections show you how to convert this Domain model into a logical ER diagram.

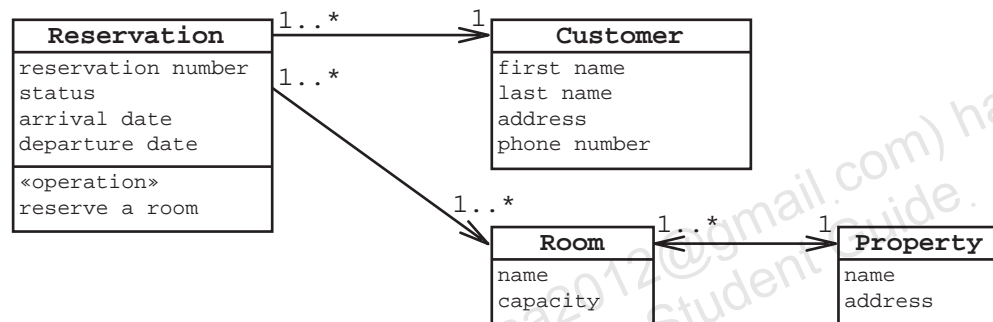


Figure 12-26 A Simplified Domain Model of the Hotel Reservation System

Step 1 – Map OO Entities to DB Tables

You can represent an ER diagram in UML by using a Class diagram. Each class node represents a table and you can use the «table» stereotype to clarify that these are database tables. You can call these “table nodes.”

In this step, create a table node for each entity in the Domain model. For example, the Hotel Reservation System includes four primary Domain classes: Reservation, Customer, Property, and Room. Figure 12-27 illustrates table nodes for these entities. Note that the data fields have been specified from the attributes of the Domain entities.

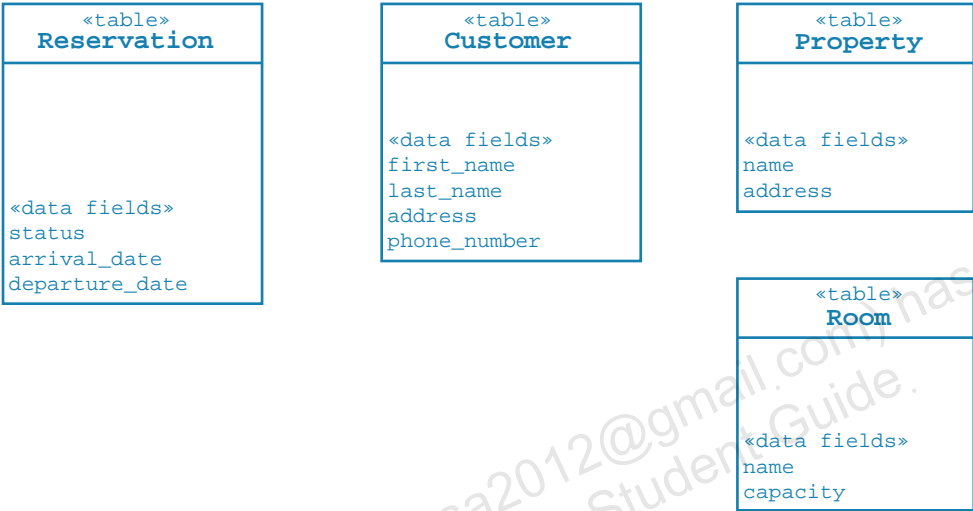


Figure 12-27 Step 1 – Creating Entity Tables

Step 2 – Specify the Primary Keys for Each Table

The primary key of a relation table is a set of fields that uniquely identify each object in the table. That is to say that a row represents a single Domain object and the primary key is the unique identifier for that object. There are two types of entities:

- Independent

An *independent entity* is one that exists independently of all other entities in the Domain model. In the Hotel Reservation System, the Reservation, Customer, and Property entities are independent.
- Dependent

A *dependent entity* is one that exists within the context of another entity. This entity usual exists in a composition relationship. For example, a hotel room exists only within the context of a hotel property. The Room entity depends upon the Property entity.

For independent entities, such as a Customer, the primary key is usually a single field. Some architects have the tendency to use a unique data field as the primary key. This is a mistake. For example, suppose that a software company is building a payroll system for the United States market. It is true that every employee in the US has a unique Social Security Number (SSN). The software company might choose the SSN as the primary key for the Employee table. What would happen if this company decided to market their software to European companies? The SSN does not exist outside of the US, so they would have to redesign their database tables with a potentially difficult schema conversion.

Therefore you should use an arbitrary key for independent entities. An integer is usually sufficient. In the Hotel Reservation System the Reservation, Customer, and Property tables are given a unique key field.

For dependent entities, such as Room, the primary key is a compound key consisting of the primary key from the parent table and a qualifier. The qualifier is a field within the dependent entity that uniquely identifies that dependent object within the parent object. For example, in the Hotel Reservation System every room has a number that is unique to that property. Therefore, the primary key of the Room table is a compound key of the `property_id` and the `room_number` fields.

Figure 12-28 shows the addition of the primary key fields.

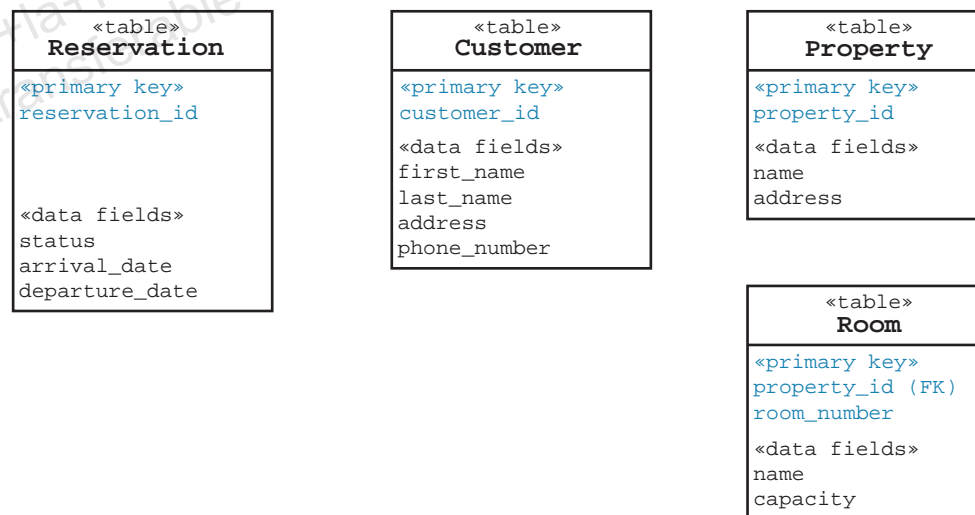


Figure 12-28 Step 2 – Specifying Primary Keys

Step 3 – Create an ER Association as a One-to-Many Relationship

Each association in the Domain model is represented by a foreign key relationship in the ER diagram. A foreign key is a field³ within a table that refers to the primary key of the related table. For example, in the Domain model a reservation is related to a single customer. This relationship can be modeled in the database with a foreign key in the Reservation table to the Customer table (using the `customer_id` key field).

In the case of dependent entities, the relationship is built into the primary key of the dependent table. For example, in the relationship between Property and Room, the Room table has the `property_id` foreign key that relates the Room table back to the Property table.

Figure 12-29 shows these relationships.

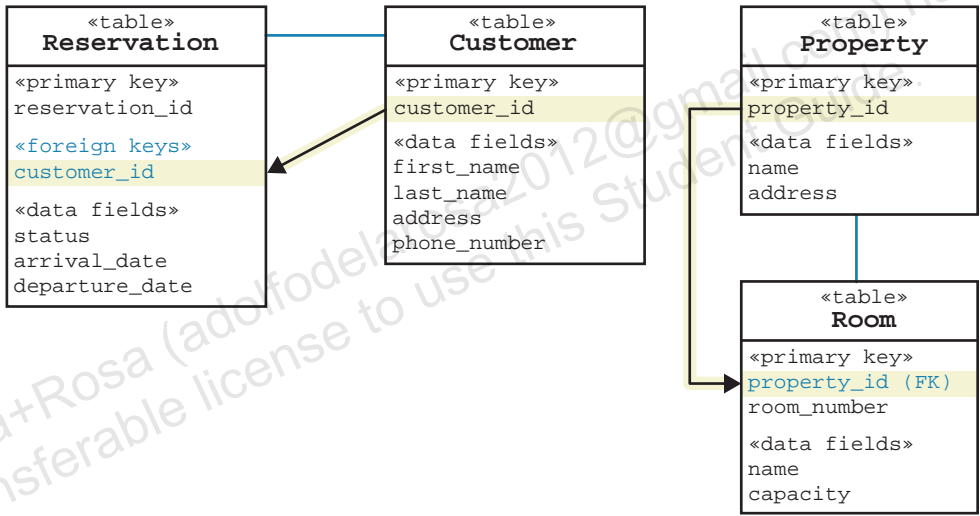


Figure 12-29 Step 3 – Creating One-to-Many Relationships

Step 3 – Create an ER Association as a Many-to-Many Relationship

3. A foreign key might be represented by multiple fields if the related table has a compound primary key.

Some object associations are many-to-many, such as the association between Reservation and Room. This means that a reservation can hold many rooms, and a single room can have many reservations (but not in the same time period). Many-to-many relationships must be modeled with a resolution table. In this example, you would create a table called ResvRooms that links the Reservation and Room tables together. Figure 12-30 shows this ResvRooms table.

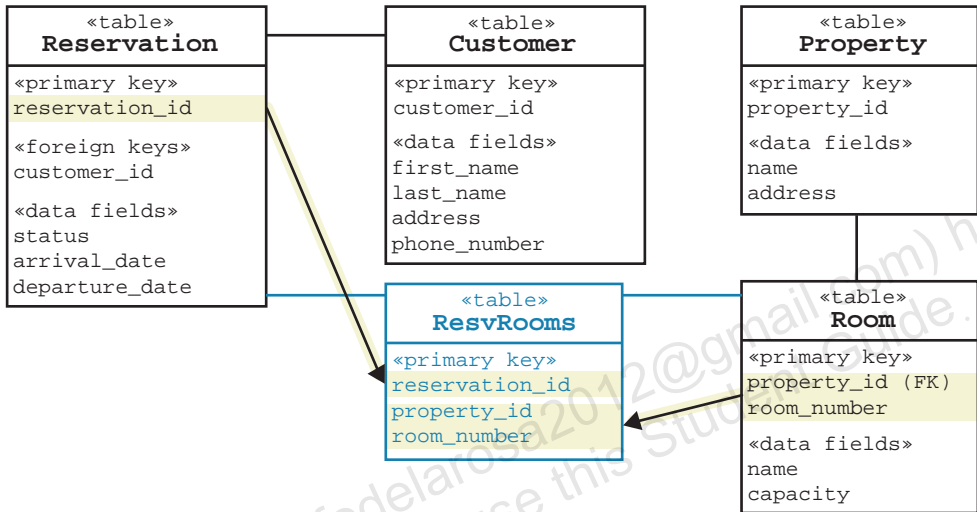


Figure 12-30 Step 3 – Creating a Many-to-Many Resolution Table

The ResvRooms table usually⁴ contains only the primary keys from each associated table. For example, reservation_id from the Reservation table and the compound key (property_id, room_number) from the Room table. This table will contain a row for each room in a given reservation, for all reservations.

4. The attributes of an association class along a many-to-many association can be stored in the resolution table.

Overview of the Detailed Deployment Diagram

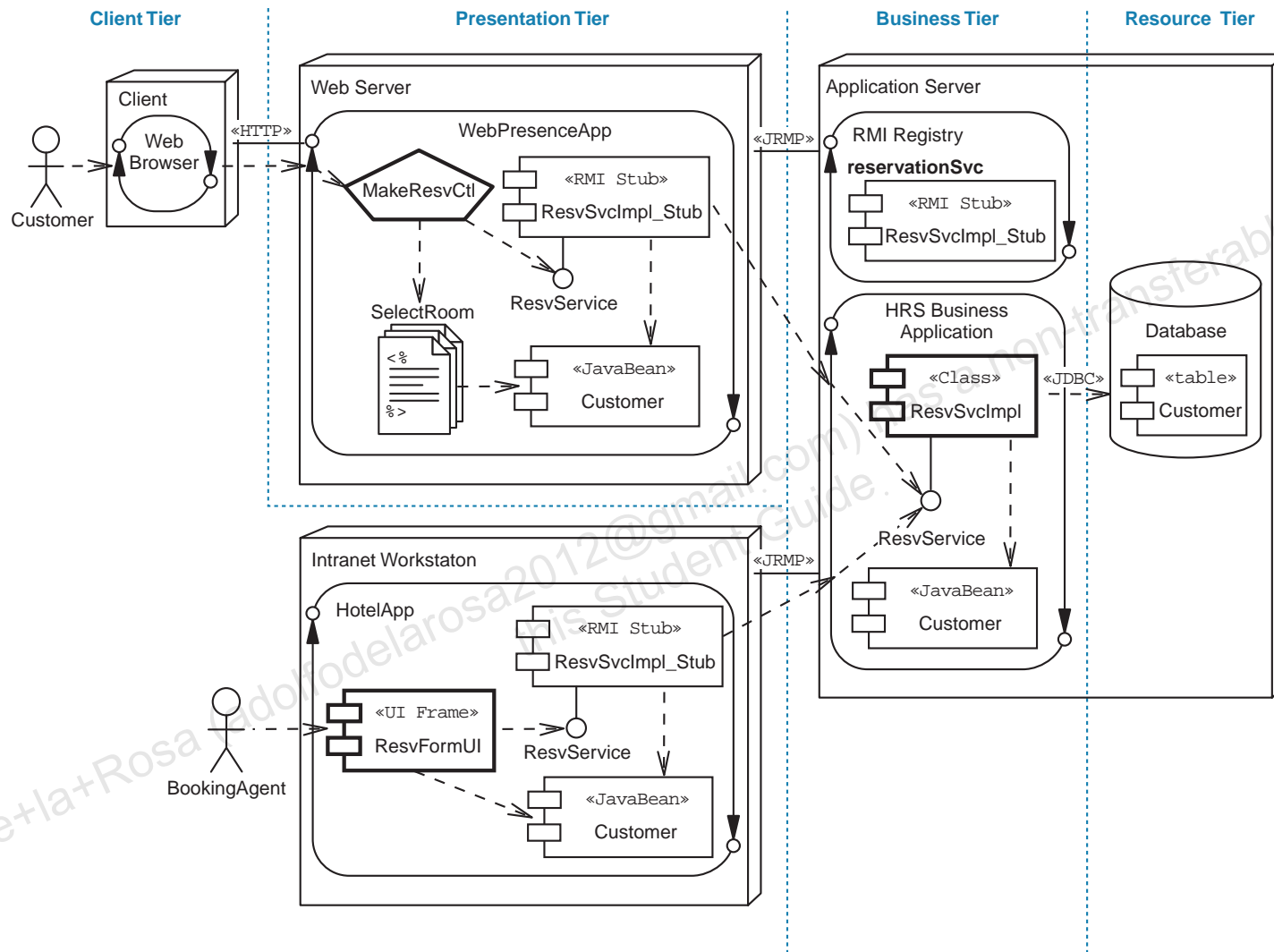


Figure 12-31 Example Detailed Deployment Diagram

Overview of the Tiers and Layers Package Diagram

Figure 12-32 shows a portion of the tiers and layers Package diagram for the HRS with the technology choices for the Resource tier filled in.

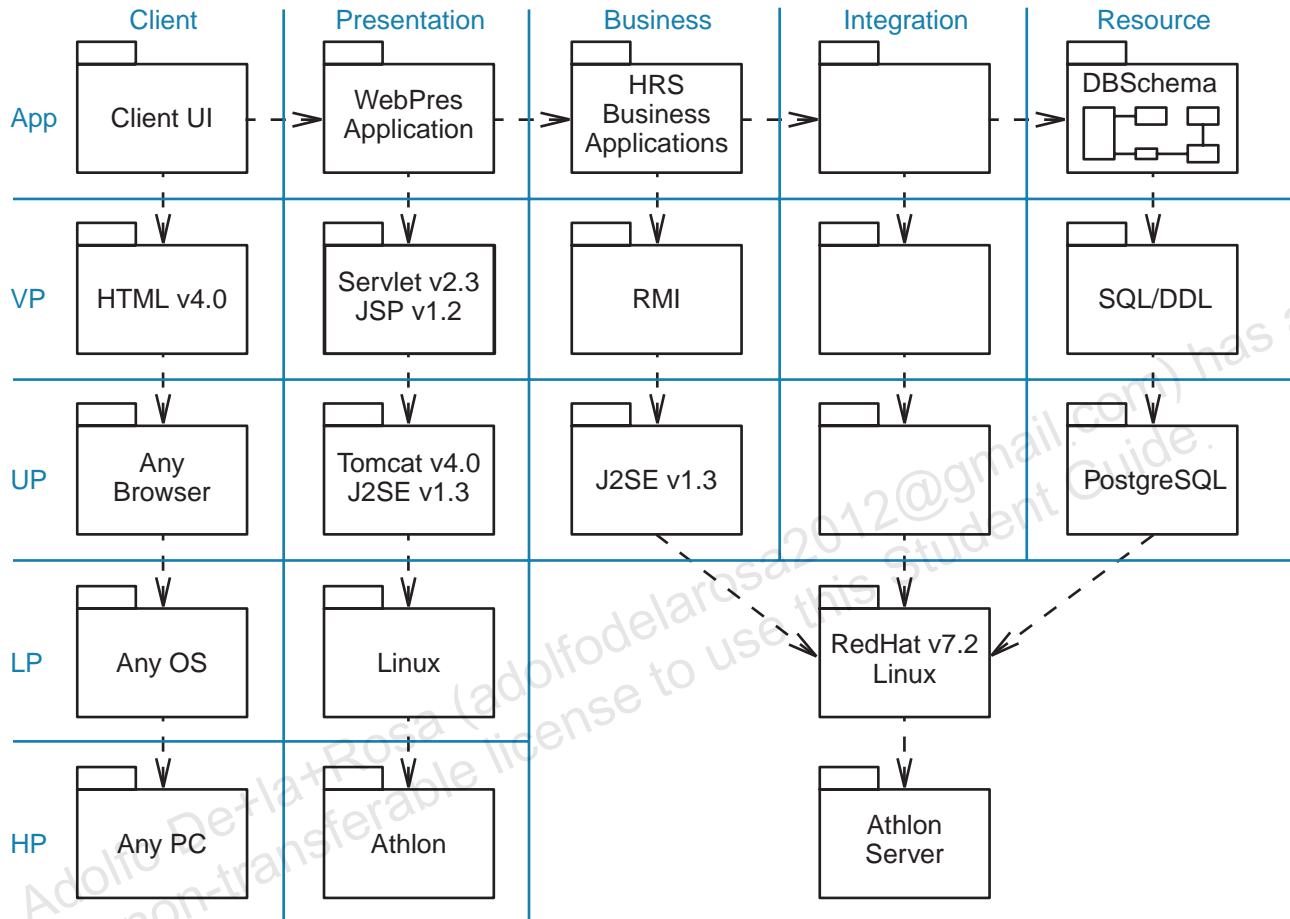


Figure 12-32 Partial HRS Tiers and Layers Package Diagram

On the Resource tier, the Hotel Reservation System is architected using a relational schema; therefore, it depends on the structured query language (SQL) specification for the data definition language (DDL). The Upper Platform for the Resource tier is the PostgreSQL DBMS server. The architect has decided to host the database on the same server workstation with the HRS Business Application. This can be represented by having the Upper Platform packages for these logic tiers to depend on a single Lower Platform and Hardware Platform package.

Exploring Integration Tier Technologies

This section explores the concepts and an architectural pattern for developing a persistence integration mechanism. You will also see how to populate the Architecture model with the design choices for the Integration tier.

The Integration tier separates the entity components from the resources.

The Integration tier in SunTone Architecture Methodology contains the set of software components that connect (or integrate) the Business tier with any and all resources in the Resource tier. Resources that require integration are:

- Data sources
A data source is a persistent object storage mechanism. As discussed in “Exploring Object Persistence” on page 12-33, there are many such mechanisms.
- Enterprise Information Systems (EIS)
An EIS refers to any pre-existing software system that provides irreplaceable value to the entire company. These might be proprietary legacy systems or commercial products such as IBM’s Customer Information Control System (CICS).
- Computational libraries
A computation library is any software system that provides mathematical or simulation capabilities.
- Message services
A message service is any tool that enables independent applications to communicate using asynchronous messages.
- Business to Business (B2B) services
A B2B service is a service that integrates two or more applications across multiple companies.

This section only discusses integration with data sources.

DAO Pattern

The DAO pattern provides the integration components that the system software uses to communicate with the data store. This pattern has the following characteristics:

- Separates the implementation of the CRUD operations from the application tier.

Applies the principle of Separation of Concerns to separate the business logic components from the data integration components. The DAO pattern supports this separation of concerns by creating components specifically designed to perform the CRUD operations. This encapsulation enables changes to the data store without affecting the Business tier.

- Encapsulates the data storage mechanism for the CRUD operations for a single entity with one DAO component for each entity.

Each DAO component hides the implementation of the CRUD operations. There is usually one DAO component for each Domain entity.

- Provides an Abstract Factory for DAO components if the storage mechanism is likely to change

The DAO pattern can provide an *Abstract Factory* of DAOs. For example, you could build DAO components for each entity for two different data stores. For example, one data store might be a set of XML files and another data store might be an RDBMS. Using an Abstract Factory enables the system to switch between the two different data stores without affecting the Business tier.

It is rare to use two different data stores (XML and RDBMS) simultaneously, but using an Abstract Factory enables the system to evolve from one data storage mechanism to another easily.

Finally, you can use the DAO pattern without using a factory. For example, if you can assume that the system's DB will always be a relational DBMS, then you do not need to implement the Abstract Factory elements of this pattern.

Details of the DAO Pattern

Figure 12-33 illustrates the DAO pattern using a Class diagram. In this example, there is an `DAOFactory` interface that provides a set of factory methods to create a DAO component for a specific Domain entity. For example, in a Car Rental domain, the `makeDepotDAO` method creates a `DepotDAO` object to handle the CRUD operations for Depot entities. There are two implementations of this Abstract Factory interface, one for XML files and one for an RDBMS. The `DepotDAO` interface provides an abstract interface to the CRUD operations for the Depot entity. There are two implementations of the `DepotDAO` interface, one for XML files and another for an RDBMS.

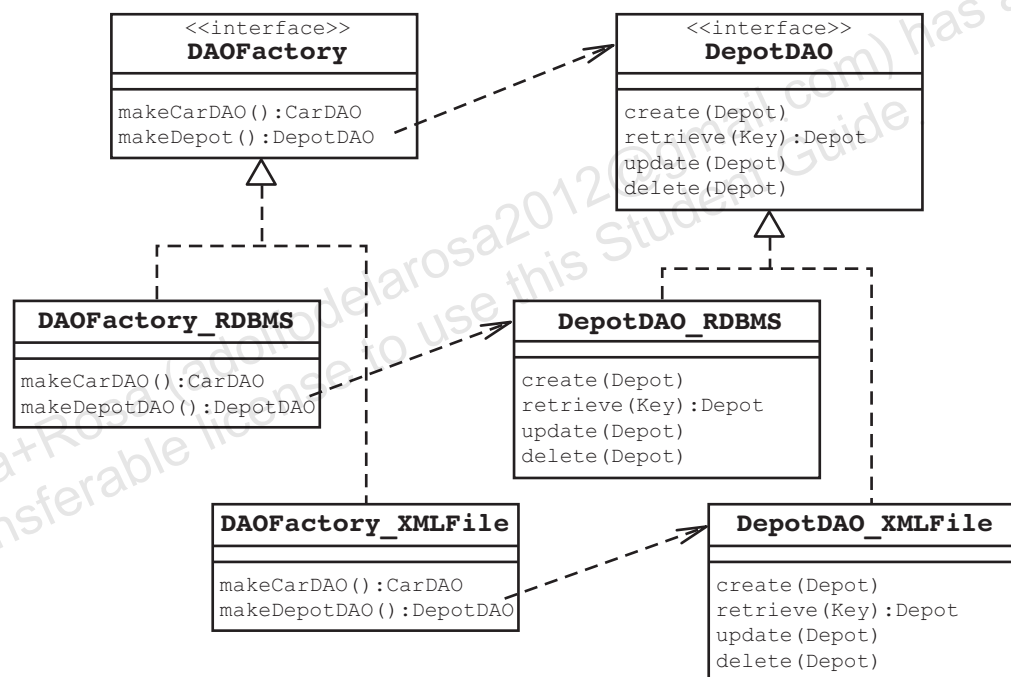


Figure 12-33 DAO Pattern Example 1

At runtime, the system will select which type of data store DAOs are needed. The business tier then uses the selected DAO factory to create the specific entity DAO components.

Figure 12-34 illustrates the DAO pattern using a Class diagram. In this example, there is a Factory method to create the Depot entity from a local database or from data accessible from a Web service. In this example, the data access for all of the objects would have to be either from the local database or from the remote Web service. Also, the local database access would be synchronous, but the Web service access would be asynchronous. This difference would have to be considered.

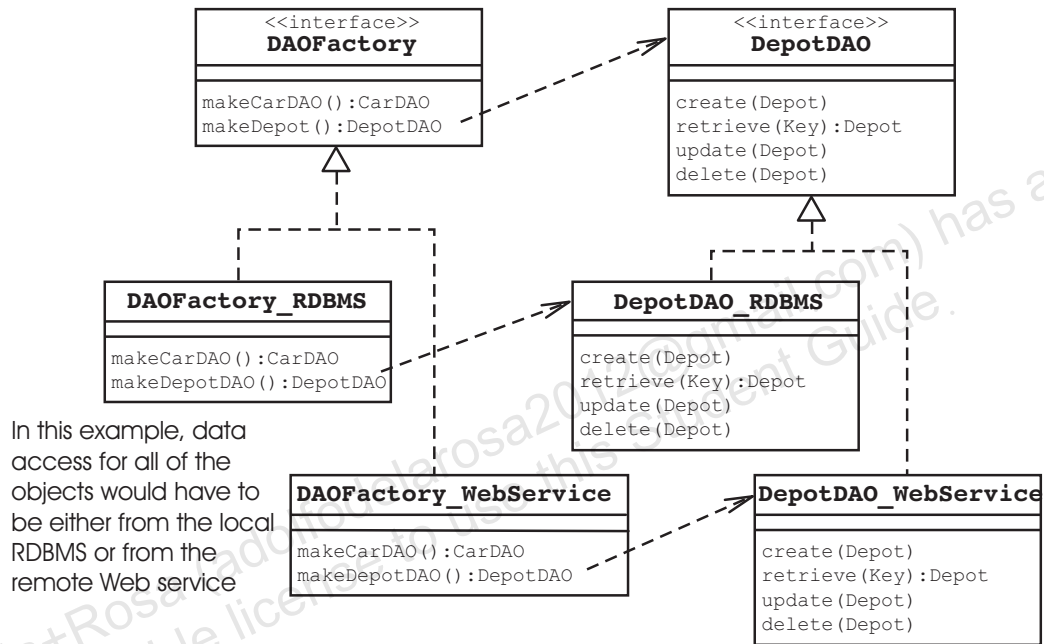


Figure 12-34 DAO Pattern Example 2

Figure 12-35 illustrates how the Business tier components depend on the Integration tier components to interact with the data store.

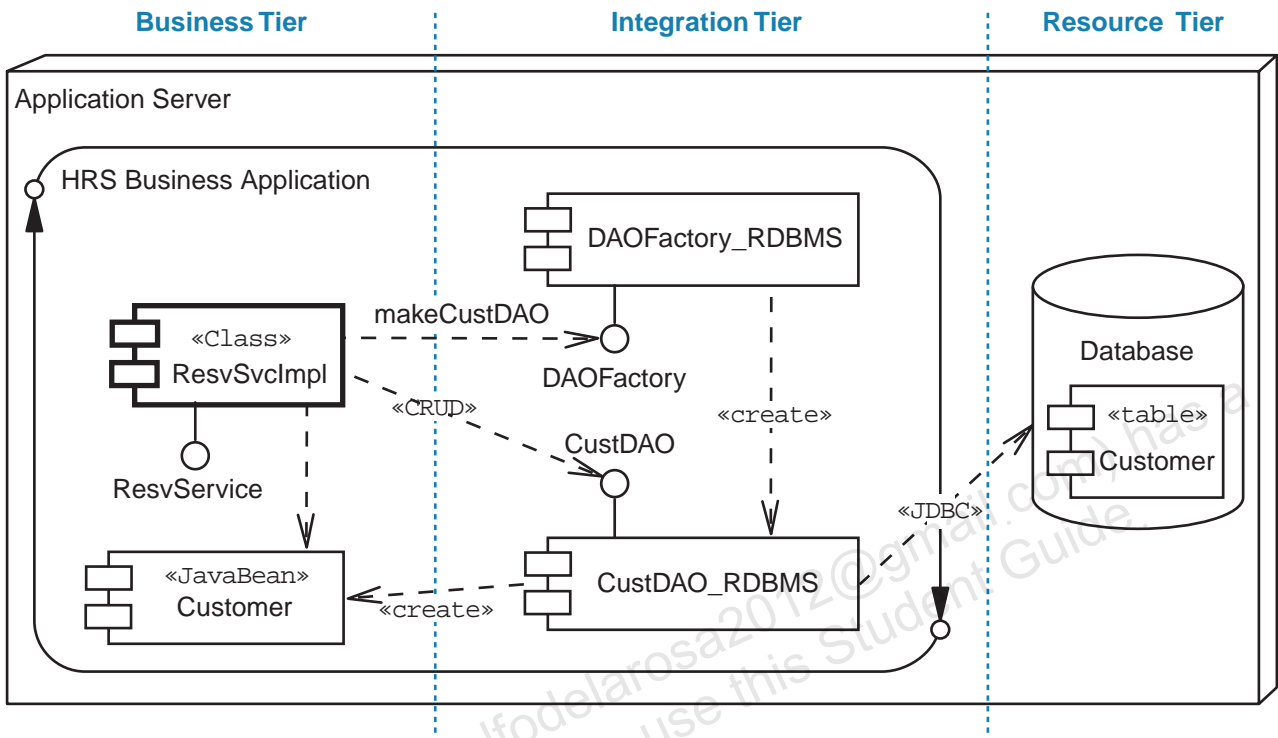


Figure 12-35 The DAO Pattern in a Detailed Deployment Diagram

In this example, the reservation service component uses the DAO factory to create a Customer DAO component. The service component then uses the DAO to retrieve a Customer object. The DAO component then communicates to the data store; in this case, using JDBC and SQL statements.

Introducing the Resource and Integration Tiers

Overview of the Detailed Deployment Diagram

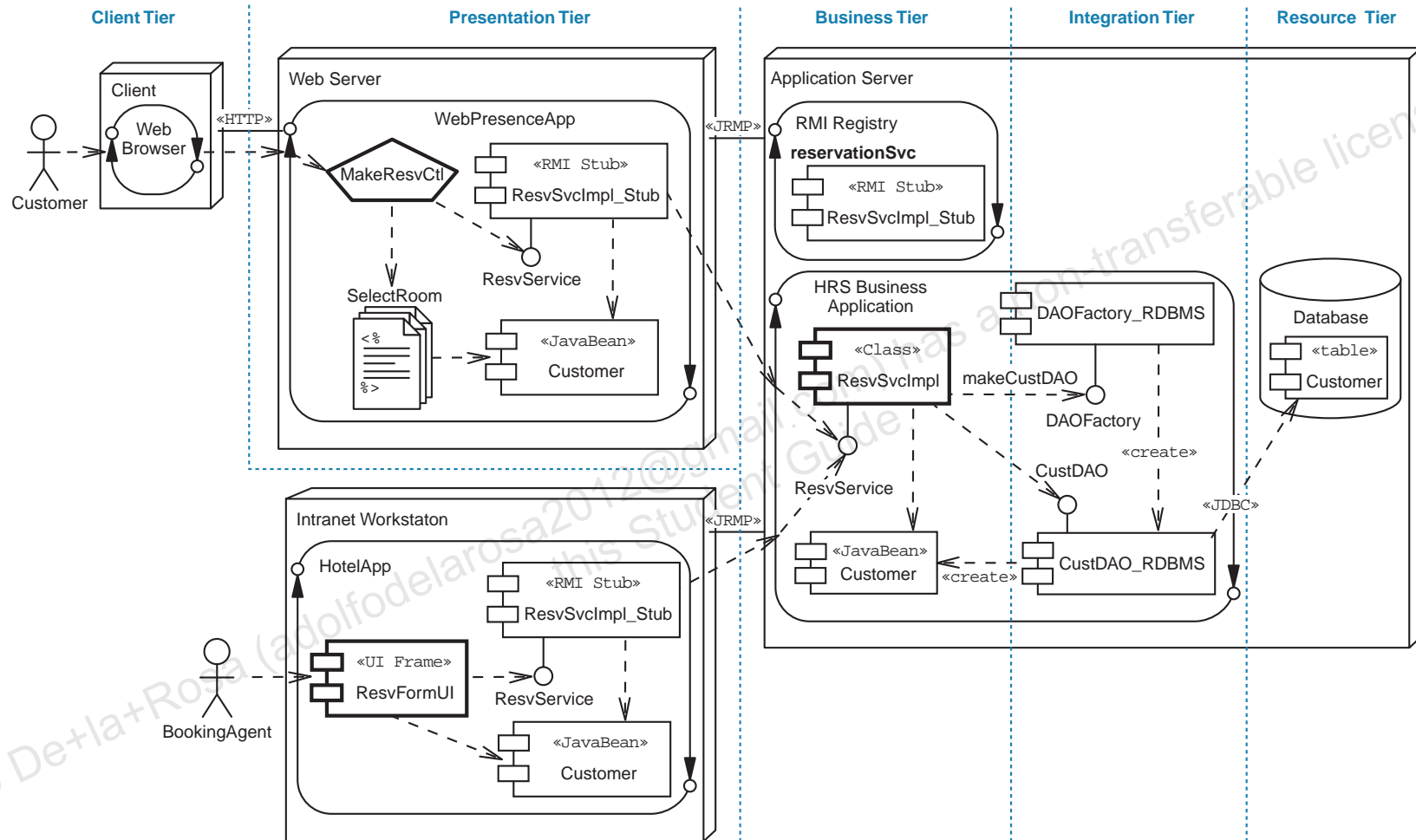


Figure 12-36 Example Detailed Deployment Diagram

Overview of the Tiers and Layers Package Diagram

Figure 12-37 shows the complete tiers and layers Package diagram for the HRS with the technology choices for the Integration tier filled in.

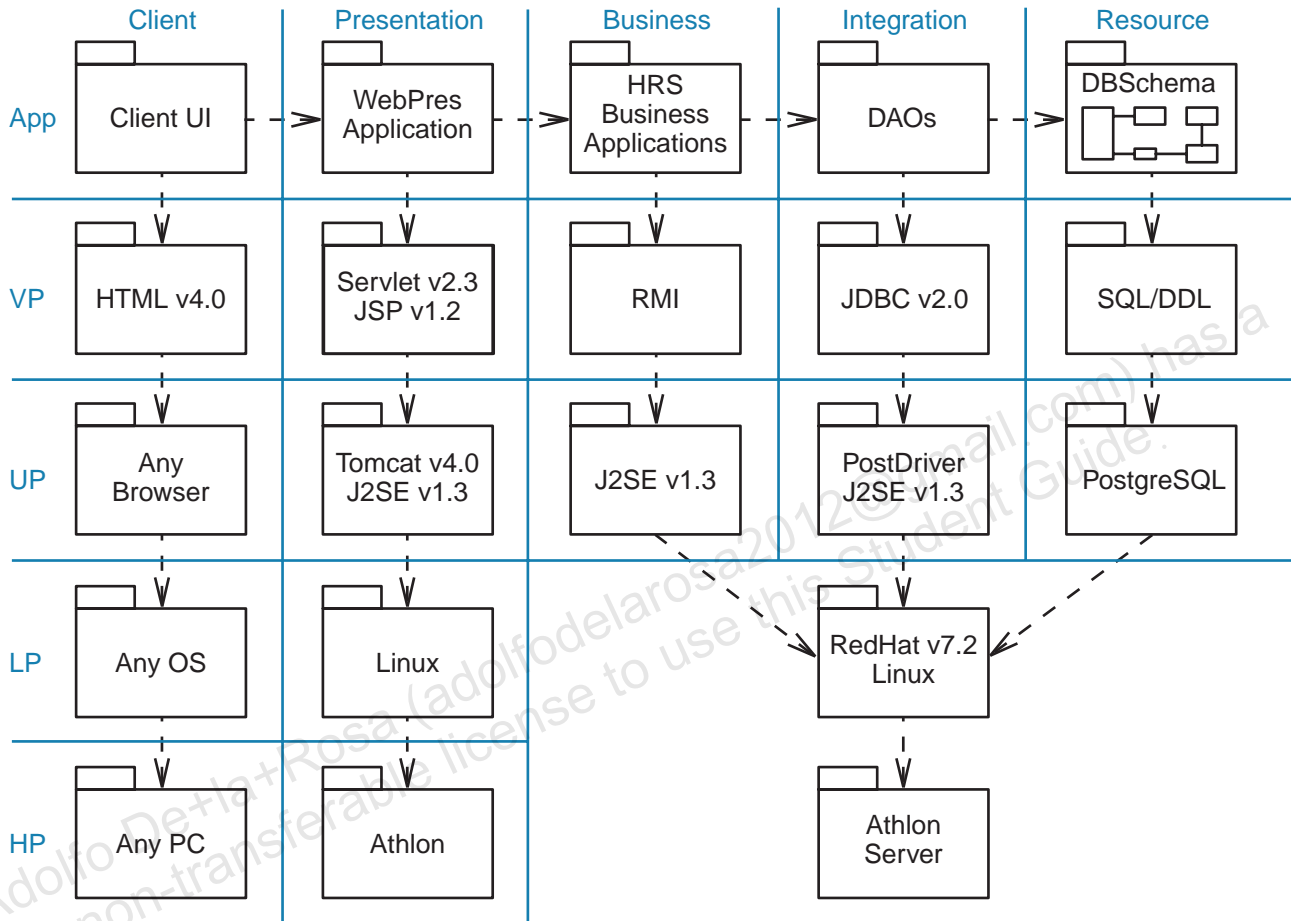


Figure 12-37 Complete HRS Tiers/Layers Package Diagram

In the Integration tier, the Hotel Reservation System is created using the DAO pattern. The DAOs are written to the JDBC interfaces. The PostgreSQL JDBC driver (abbreviated PostDriver) is the library that communicates to the database.

Java™ Persistence API

Java Persistence API:

- Is an alternative to DAO

Although the Java Persistence API is an alternative to using DAOs, it may be possible to use legacy DAOs to use the Java Persistence API. Therefore, you do not have to change your existing business logic components.

- Draws on the best ideas from alternative persistence technologies such as Hibernate, TopLink, and Java Data Objects (JDO)
- Uses a persistence provider such as Hibernate or TopLink
- Is a Plain Old Java Object (POJO) persistence API for Object/Relational mapping

The Java Persistence API does not need to exist within a Java EE container, so it may be used with POJOs.

Note – Object to Relational mapping may be defined with Java annotations added to the Java class source file.

- Uses an entity manager to manage objects

Java Persistence API Entity Manager

A Java Persistence API entity manager can be requested to:

- Manage objects by keeping their data in synchronization with the database record
- Persist or merge unmanaged objects, which makes these objects managed
- Find an object using the primary key, which creates an object from the database record

The Java Persistence API or any of the comparable technologies provide a more transparent link between an object and the database. This is because object modifications for managed objects do not need the business logic to actively save the changes. However, this may have a negative effect on performance.



Introducing the Solution Model

The Solution model is the basis upon which the development team will construct the code of the system solution. Figure 12-38 shows the Solution model is constructed by merging the Analysis model into the Architecture model (template).

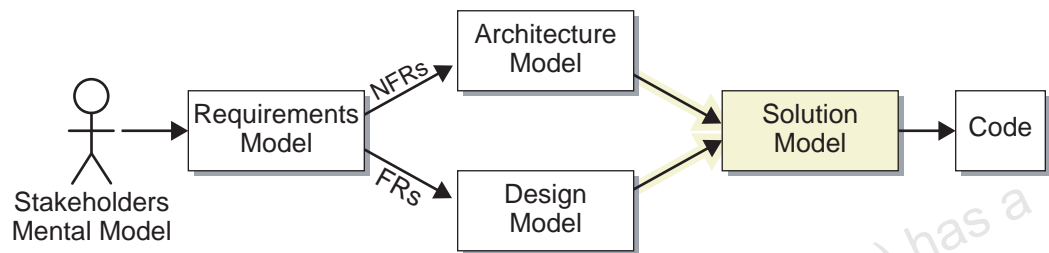


Figure 12-38 Introducing the Solution Model

Conceptually, the Solution model represents the primary software components that solve a use case (modeled by the Analysis model). Because this model is very close to the actual code that will be developed, these models can be quite large. It is not always a good idea to develop a complete Solution model, but this model can be invaluable to a novice development team.

A Solution model can be represented by actual code; for example, if the architecture team built an Architecture baseline. The structure of this code set can act as a Solution model for the development team.

In this course, the Solution model is represented by a detailed Deployment diagram.

Overview a Solution Model for GUI Applications

The Analysis model for GUI applications can be modeled with the standard set of Analysis component types. Table 12-1 shows the icons for these components.

Table 12-1 Standard Analysis Component Icons

Component Type	Icon
Boundary	
Service	
Entity	

To create a Solution model, you must have a Analysis model. Figure 12-39 provides an example Analysis model from the Hotel Reservation System.

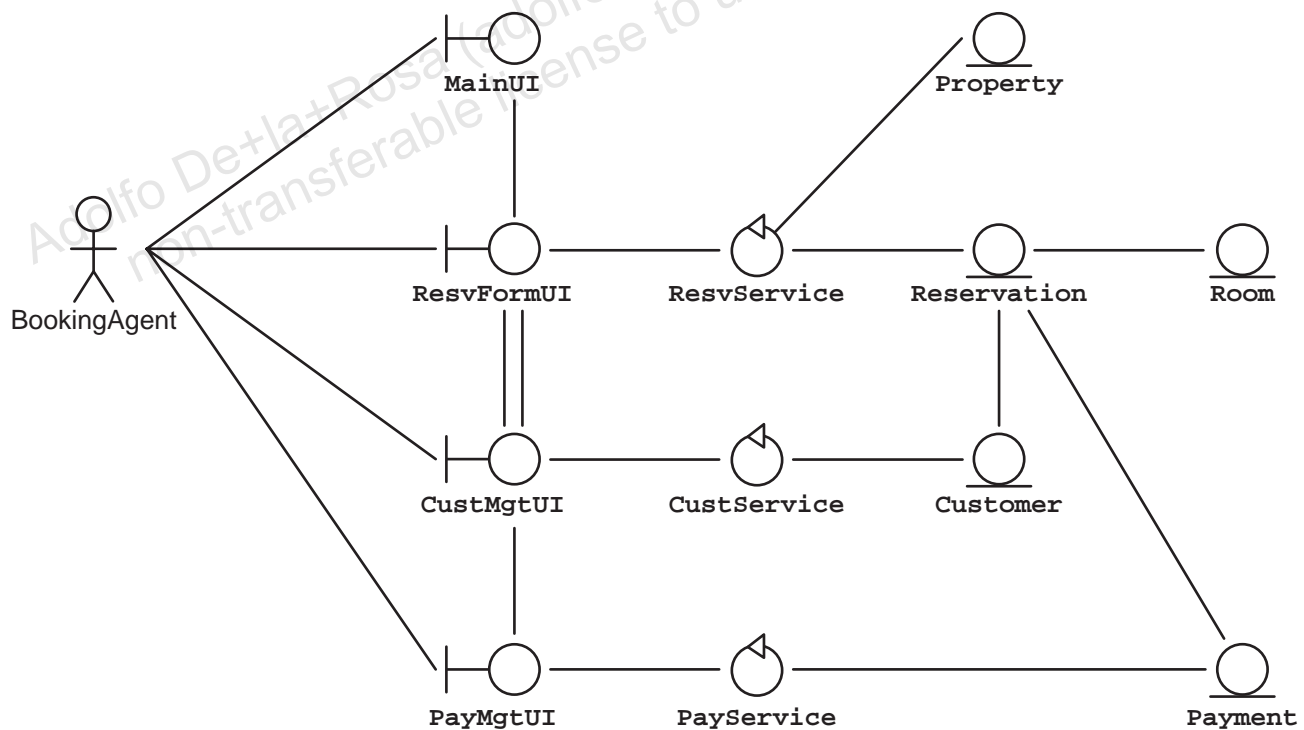


Figure 12-39 A Complete Analysis Model for the Create a Reservation Use Case

Using the PAC pattern, the reservation agent has three primary subcomponents, the ResvUI (presentation), the Reservation (abstraction), and the ResvService (control). The reservation agent also makes use of two additional agents, customer management agent and the payment agent.



Note – This diagram only shows the Analysis components and their relationships. It does not show the collaboration messages because these are not needed for the creation of the Solution model.

Figure 12-40 on page 12-54 shows the Analysis model components distributed across the architectural tiers.

Introducing the Solution Model

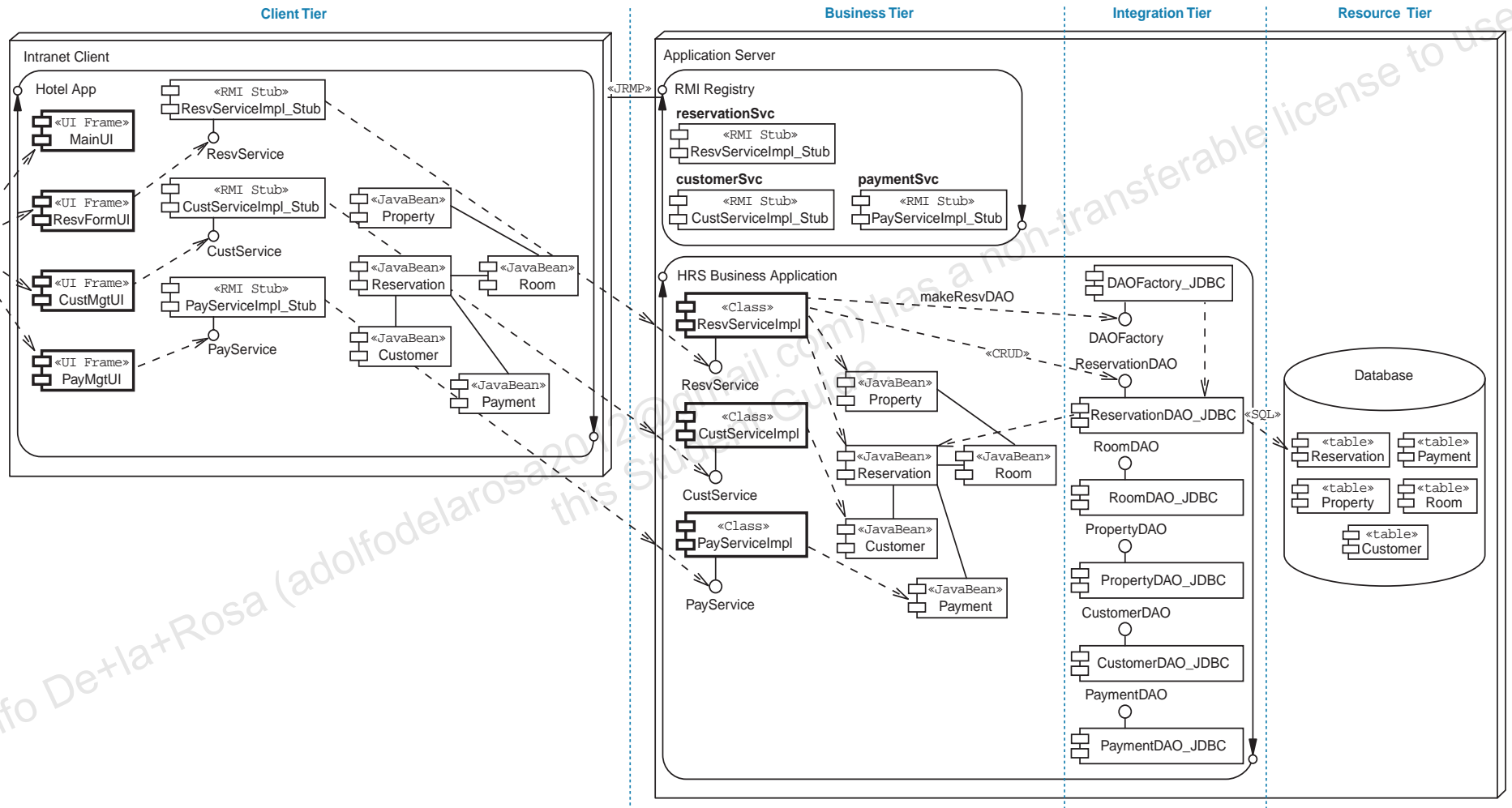


Figure 12-40 A Complete Solution Model for the Create a Reservation Use Case

Overview a Solution Model for WebUI Applications

The Analysis model for Web User Interface (WebUI) applications cannot be modeled adequately with the standard set of Analysis component types, because Web applications (using Java technologies) do not have a single component that maps directly to a Boundary component. Instead, Web applications have Controller and View components. Table 12-2 shows the icons for these components.

Table 12-2 WebUI Analysis Component Icons





Component Type	Icon
View	
Controller	
Service	
Entity	

Figure 12-41 provides an example Analysis model for a Web application.

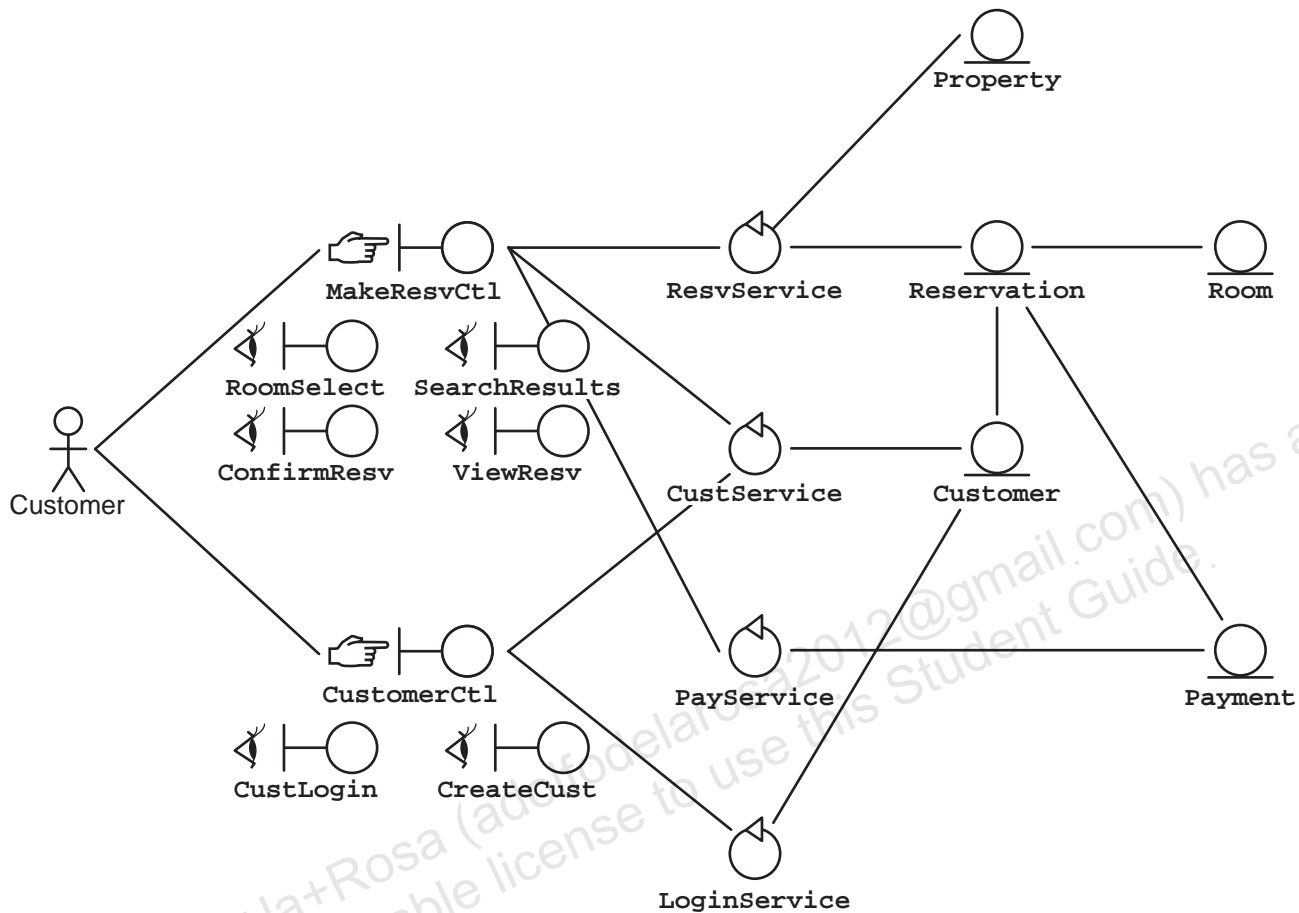


Figure 12-41 A Complete Analysis Model for the Create a Reservation Online Use Case

In this example, the online reservation software uses the same set of Service and Entity components, with the addition of the LoginService component. The main difference is the structure of the user interface. In this web application, the software uses a primary Controller servlet, ResvCtl, with the JSP technology page views for creating a reservation. There is also an auxiliary servlet controller, CustCtl, which enables a user to create a new customer account or to log in as an existing customer

Figure 12-42 on page 12-57 illustrates a complete solution model for the “Create a Reservation Online” Use Case.

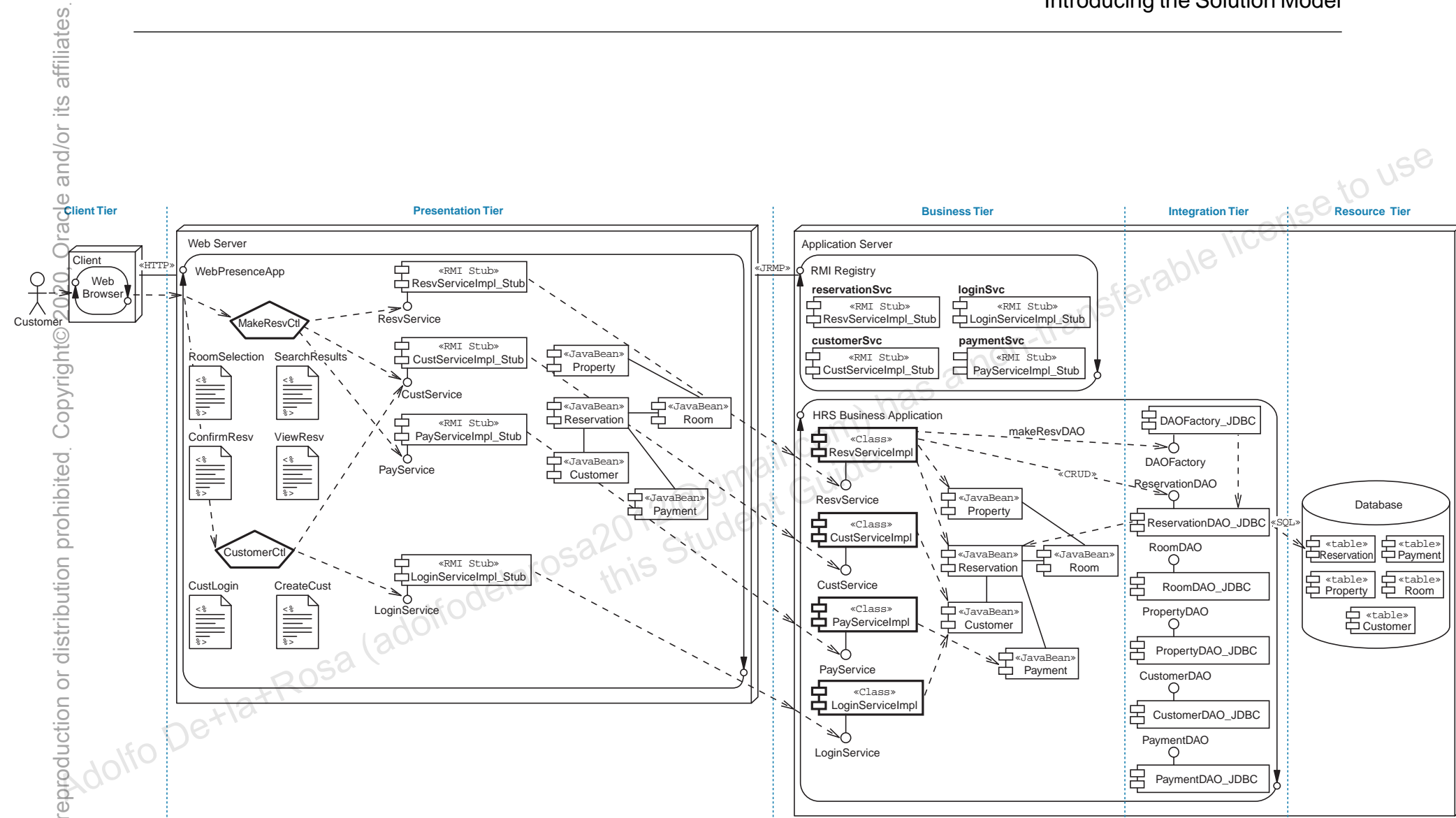


Figure 12-42 A Complete Solution Model for the Create a Reservation Online Use Case

Summary

In this module, you were introduced to the architectural tiers. Here are a few important concepts:

- The Client and Presentation tiers includes Boundary View and Controller components.
- The Business tier includes business services and entity components.
- The Integration tier includes components to separate the business entities from the resources.
- The Resource tier includes one or more data sources such as an RDBMS, OODBMS, EIS, Web services, or files.
- The Solution model provides a view of the software system that can be implemented in code.
- The Solution model is created by merging the Design model into the Architecture model (template).