Introducing Architectural Concepts and Diagrams

Objectives

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

- Distinguish between architecture and design
- Describe tiers, layers, and systemic qualities
- Describe the Architecture workflow
- Describe the diagrams of the key architecture views
- Select the Architecture type
- Create the Architecture workflow artifacts

Additional Resources

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

- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Upper Saddle River: Addison Wesley Longman, 1998.
- 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.
- Tarr, Peri. "Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001." [`http://www.research.ibm.com/hyperspace/workshops/icse2001/`], accessed 11 October 2002.
- The Object Management Group. "Unified Modeling Language (UML), Version 2.2" [`http://www.omg.org/technology/documents/formal/uml.htm`].

Process Map

This module describes some of the diagrams created by the Architect job role. Figure 11-1 shows the activities and artifacts of the Architecture workflow.

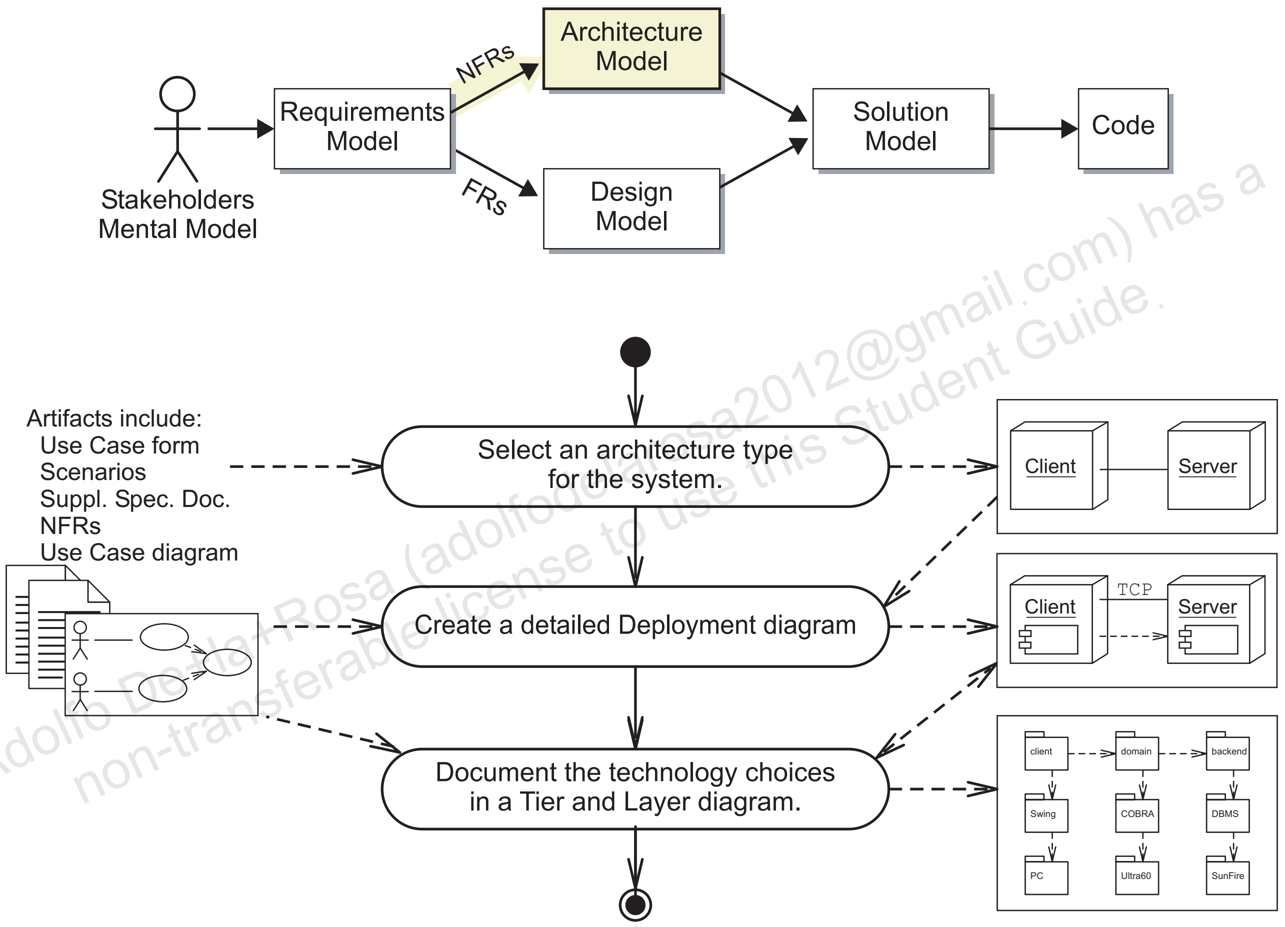

Figure 11-1 Architecture Process Map

Justifying the Need for the Architect Role

For years, software has been developed without a special role called *Architect*. Why is it that software engineering groups are now employing people in this role?

In recent years, two crucial changes have occurred in software engineering. The first, and most obvious, is one of scale. Scale (for example, how many users) has been increasing since the first computer program was written. In web applications, the scale of systems is a critical issue because you can never know exactly how many users will be requesting services.

The second change, which has been much more sudden, is the distribution of software components across multiple servers. For example, web applications are distributed by nature. The huge increases in load that are typical of Internet-deployed systems tend to make multiple parallel servers necessary, and this further increases distribution.

So, why is distribution a reason to create a new role in software engineering, and why is it appropriate to call this role *Architect*? The answer to this question is the focus of this module.

Risks Associated With Large-Scale, Distributed Enterprise Systems

As systems move from a single host to large-scale, distributed enterprise systems, the risks and difficulty of meeting project requirements increases.

Minimally Distributed Systems

In a system that runs entirely on a single host, all pieces of the system communicate with each other reasonably quickly and at approximately the same speed. Even when the system becomes a client-server system (Figure 11-2), communication is reasonably easy to control. This situation is the premise of the saying “compute near the data; validate near the user.”

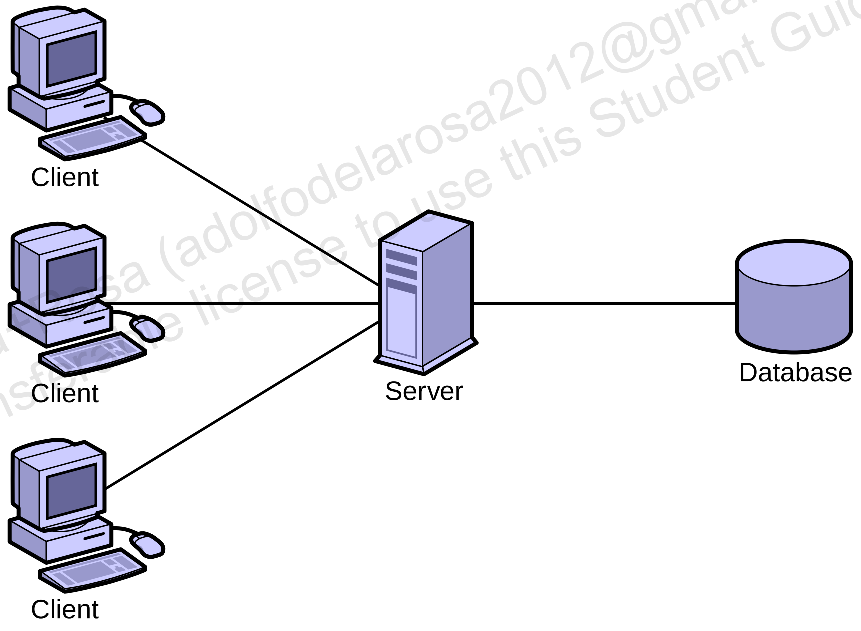

Figure 11-2 Client-Server System

Highly Distributed Systems

In a more distributed system (see Figure 11-3 on page 11-6), making the right choices about where components are located and how they communicate becomes much more critical and difficult. The importance of these decisions created the need for architects.

The critical role of the architect is to perform high-level planning for the location of, and the communication among, software components. This role distinguishes the architect from the designer, programmer, integrator, and others.

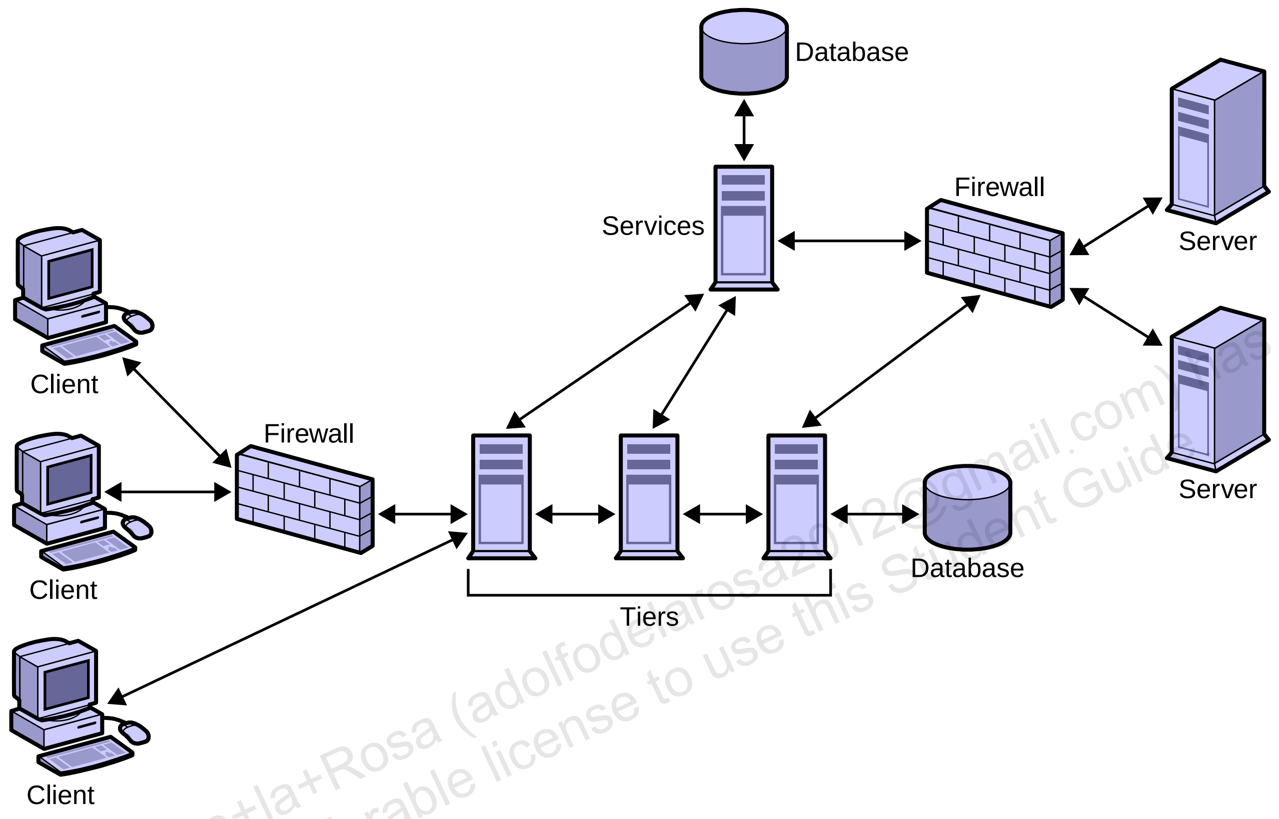

Figure 11-3 Highly Distributed Systems

The goal of this high-level planning is to ensure that the system:

- Is robust if partial failures occur
- Handles the required load
- Is scalable when the demand for concurrent use exceeds the original design parameters

Note – This course cannot explore the Architecture workflow in too much depth due to the complex nature of the tasks. However, the next module "Introducing the Architectural Tiers"" in Module 12, "provides an overview of a few of the Architecture artifacts and how the Architecture template fits into the Design workflow (see . Also, Sun Learning Services provides a five-day instructor-led course on architecture: *Developing Architectures for Enterprise Java™ Applications* (SL-425).

Quality of Service

The architect is primarily concerned with quality of service: ensuring that the performance and reliability of the system are high enough that the users are willing to use the system. The architect works with use cases and provides crucial input to the architectural development. But use cases are not the architect's main focus.

Project Failure

Project failure is not usually attributable to functionality problems. Many projects reach functional completeness, but they are so delayed or otherwise unacceptable to the end users that they are abandoned. Attention to systemic qualities can help you to avoid these pitfalls and can lead you to project success.

Non-Functional Requirements

The focus on non-functional requirements is generally considered to be the primary purpose of architecture. This focus has become necessary because a highly distributed system mandates a good architectural design to ensure that the system performs sufficiently well when all the functional requirements are met.

Non-functional requirements can be grouped into the following categories:

- Constraints

  The limits on the decision making that can be done while designing the system. Examples include the use of HTTP for communication with clients or the use of an existing database management system.

- Systemic qualities

  The requirements that guarantee a certain quality of service can be achieved after the system is deployed. Systemic qualities include scalability, performance, availability, and reliability, among others.

Generally, the focus on quality of service is a surprise to many experienced designers. They often find focusing on quality of service to be frustrating, and they want to work with more tangible design aspects. However, experience is proving that as long as a system has a good architecture, other problems are fixable. By contrast, the system lacking a good architecture might be functionally perfect, but still require a total rewrite to be usable or to revise the functionality.

Risk Evaluation and Control

A good architect controls risk to guarantee that the non-functional requirements are met by the system design. This means risk control is a project cost, and it must be treated as such.

Risk Evaluation

To make qualified decisions about how much risk control is enough and how much is too much, you evaluate and compare the risks and mitigation strategies based on cost and probability of occurrence. You do not want to create high-cost solutions to mitigate low-probability risks. If designing such a solution was your goal, you could simply design systems that provide T1 connections and mainframes for every client that connects. This is not the best solution, and an architect should not take a position of *throw more hardware at it... that will fix all the problems.*

Cost Analysis

The best way to approach the problem is to perform a cost analysis of several options for controlling risk. Some options might be less costly, but leave some risk in place; others might be more costly, but control the risk completely. You can always look at the *do nothing* option: assume that the risk is realized and examine the cost impact. This *pay me now, or pay me later* approach to deciding what architectural solutions fit best is formalized in the return on investment (ROI).

For example, consider the use of a single server to provide availability compared to the implementation of the system on a server cluster. The cost of implementing a single-system environment is significantly less than that of the cluster: usually three to five times cheaper, possibly even more. This savings is due to the number of systems required to create a cluster (two to five or more, depending on configuration and types of systems used) and the level of expertise required to install and maintain the configuration.

On the other hand, you must consider the cost of lost time and business in the single-system configuration. If the system goes down, how long would it take to replace it? How much money could be lost while the system is being replaced? And how often is the single system likely to fail? This analysis might indicate that, even though the cost of implementing and maintaining a cluster might appear to be more costly at first, the long-term cost of not using a cluster might justify its use.

The Role of the Architect

The role of the architect includes technology and management responsibilities.

Technology Responsibilities

The architect must identify *architecturally significant use cases*. "Architecturally significant" describes aspects of the system that have the highest degree of risk associated with them. Perhaps the single most important aspect of an architect's work is preparing for the unexpected.

The architect is responsible for guiding the development of the architectural prototype. This prototype might be a purely theoretical construction, realized only in documents and diagrams. Or, more commonly, it might involve an element of implemented code. The purpose of the prototype is to develop enough of the final system to determine that the key risks have been successfully addressed. In developing the architectural prototype, the architect creates:

- List of assumptions and constraints

  The architect documents all the assumptions made, so far as they have been recognized. Incorrect assumptions about potentially critical issues, such as user-base growth and transaction volumes, often require the system to be reworked. Documenting these assumptions ensures that you realize as quickly as possible that something is changing that might require such rework.

  The architect documents all the constraints about the platform upon which the software is constructed and deployed.

- Risk identification and mitigation plan

  The architect provides a list of known risks, an assessment of the risks, and mitigation plans intended to address the risks.

- Deployment environment description

  The architect specifies the deployment environment and shows how and where software and service components should *be located and how those components should communicate.*

- Interaction diagrams

The architect uses these diagrams to verify that the architectural prototype supports all the functional requirements. This view of the architectural prototype ensures the domain model is complete enough to enable the architect to map elements of the domain model correctly and to complete the closure of the model.

Management Responsibilities

At a minimum, the architect must accept some management responsibilities. The exact nature of the skills required varies according to the particular project and company. The responsibilities generally include some degree of cost management, because other budget holders are not qualified to make decisions about the technologies and risk mitigation approaches.

The architect needs significant *soft* skills for working effectively with stakeholders and team members. Excellent interpersonal skills are required for tasks such as:

- Convincing other stakeholders of the validity of the decisions that have been made

  It is likely that many of the stakeholders have preconceived ideas about what technologies should be used in what parts of the system. Many stakeholders tend, quite naturally, to overvalue certain parts of the system that relate to their needs, while simultaneously undervaluing other parts of the system. The architect must smooth over such issues, and ensure that all parties are willing to accept the necessary compromises; otherwise, the resulting disagreements can damage the entire project.

- Mentoring other team members in the proper use of the technologies that have been chosen to implement a solution

  This task typically falls to the architect because of the lead role that the architect takes in selecting technologies. To ensure the technologies are applied correctly, especially when they are new or cutting edge technologies, the architect oversees the prototype development and uses it as a training tool for lead designers and developers. After they are familiar with the technologies, the other team members can be confidently relied upon to complete the remainder of the project during the Construction phase.

Distinguishing Between Architecture and Design

The role of the architect is sometimes hard to distinguish from that of designer. Indeed, the role of an architect is difficult to define. In part, this is because the role varies from company to company and even from project to project. However, some important generalizations about the architect's role can be drawn from investigating the common elements of a number of projects. These observations and experiences have been incorporated into the architect's role as defined by Sun Professional Services^{SM} Solutions services group. Table 11-1 shows how architects differ from designers.

Table 11-1 Difference Between Architects and Designers

	Architect	**Designer**
Abstraction level	High/broad Focus on few details	Low/specific Focus on many details
Deliverables	System and subsystem plans, architectural prototype	Component designs, code specifications
Area of focus	Nonfunctional requirements, risk management	Functional requirements

Note – A component is a unit of software; it need not be a class.

Although business analysts, system architects, and system designers have distinct roles, the architect often performs many of their functions. This helps to explain the difficulty of defining the architect's role. In different companies or projects, the architect role might be quite different, depending on exactly which of these additional roles the architect fills.

The architect is not responsible for the project over its entire life cycle. Instead, the architect is primarily involved in the project at the point where its life cycle is moving from the definition-of-requirements into the detailed-design phase. This occurs in the Inception and Elaboration phases. However, the architect must ensure that the final product conforms to the original vision of the Architecture model. Some projects request that the architect attend project meetings or critical design reviews throughout the Construction phase.

Architectural Principles

Architecture principles are axioms or assumptions that suggest good practices when constructing an system architecture. These principles form the basis of many architectural patterns (see "Architectural Patterns and Design Patterns" on page 11-14) and form the decisions an architect makes to satisfy the quality of service requirements.

There are potentially hundreds of architectural principles. However, the next two modules will use the following principles:

- Separation of Concerns

  This principle tells the architect to separate components into different functional or infrastructure purposes. For example, it is important to separate the user interface from business logic (called the Model). Even user interfaces can be separated into visual elements (called a View) and user interaction logic (called a Controller). This separation of concerns leads to the Model-View-Controller (MVC) architecture pattern.
- Dependency Inversion Principle

  "Depend upon abstractions. Do not depend upon concretions." (Knoernschild page 12)

A client component that depends on an interface makes the software more flexible because the supplier component can be changed or replaced without affecting the client component. This principle is illustrated in Figure 11-4.

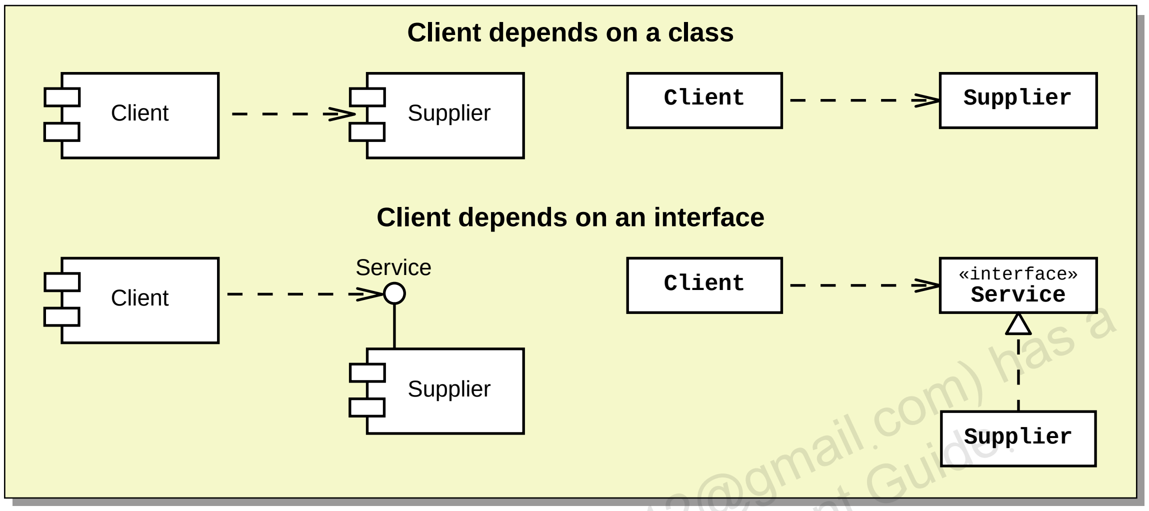

Figure 11-4 Dependency Inversion Principle

- Separate volatile from stable components

  As a system changes, you want to reduce the changes to a small number of packages. This principle guides the grouping of components into volatile elements (such as the user interface) and more stable elements (such as the domain entities).

- Use component and container frameworks

  A component is a software element that is managed by a container. For example, a servlet is managed by a web container. Typically, a software developer creates components. These components are built to fulfill interface or language specification.

  A container is a software framework that manages components built to a certain specification. For example, a web container is a framework for managing HTTP requests and dispatching the requests to a servlet. A container is a system (or library) that you can build or acquire. For example, Tomcat is an implementation of a web container.

  A container framework is a powerful tool for the architect. These frameworks enable the architect to purchase software that provides infrastructure for the rest of the software system. This enables the designers and programmers to concentrate on the business logic components rather than the infrastructure.

- Keep component interfaces simple and clear

  The more complex a component's interface, the harder it is for software developers to understand how to use the component. Component interfaces should also be highly cohesive.

- Keep remote component interfaces coarse-grained

  Remote components require network traffic to communicate. You should keep the number of remote requests (which requires sending network messages) to a minimum. Therefore, keep remote interfaces simple and coarse-grained. For example, a `LoginService` component could have three methods: `setUserName`, `setPassword`, and `performLogin`. However, this would require three network messages to perform a single login operation. Instead, the `LoginService` component could have a single method: `login(username, password)`. Even though both techniques send approximately the same amount of data, the latter technique incurs the latency of only the one network message rather than three.

Architectural Patterns and Design Patterns

Patterns are standard solutions to commonly recurring problems in a particular context. By using a pattern-based reasoning process to plan systems, the architect can analyze systems to identify problems and trade-offs with respect to service-level requirements. Each of these problems might be solved by applying a particular pattern, or it might require further analysis and decomposition. The architect can then synthesize the system by combining the pattern solutions into aggregate constructs.

To be effective at selecting and applying patterns, an architect must be familiar with a variety of pattern catalogs. Each of these catalogs focuses on a particular aspect of system development: analysis, design, architecture, and technology.

The architect primarily uses design and architectural patterns:

- Design patterns define the structure and behavior to construct effective and reusable OO software components to support the functional requirements.

  For example, one commonly used design pattern is the Composite pattern. This pattern supports the ability to treat collections of objects in the same manner (using the same methods) as the individual objects. For example, a computer manufacturer might have an inventory system that calculates cost of individual components, such as memory sticks, CPUs, hard drives, and so on, as well as complete system, which are a composite of individual components. This is a functional requirement.

- Architectural patterns define the structure and behavior for systems and subsystems to support the nonfunctional requirements.

  For example, one commonly used architectural pattern is the Layers pattern. This pattern describes a technique for partitioning systems into abstractions in a way that allows coupling only between adjacent abstractions. Common examples of layering include APIs, virtual machines, and client-server structures.

  Another common pattern is the Pipes and Filters pattern. This pattern is based on the idea that data moves from host to host and undergoes some value-added processing within each host. A data transfer path is a pipe, and a data processor is a filter. Examples of this pattern include UNIX® tools and many common compilers.

The relationship between architectural patterns and design patterns is basically one of scale and focus. Architectural patterns specify systems in terms of subsystems and high-level components that fulfill service-level requirements. Design patterns are used when creating the individual subsystems and lower-level components that make up the subsystems specified by an architectural pattern. Design patterns directly support the functional requirements.

Note – You use both architectural and design patterns in a similar way. To build large, complex constructs, you usually apply several patterns to achieve the structure and behavior that you are looking for.

Tiers, Layers, and Systemic Qualities

The SunTone Architecture Methodology recommends the following architectural dimensions to consider:

- The tiers to separate the logical concerns of the application
- The layers to organize the component and container relationships
- The systemic qualities identify strategies and patterns across the tiers and layers

These three dimensions are often represented graphically by a 3D cube diagram. Figure 11-5 shows this diagram.

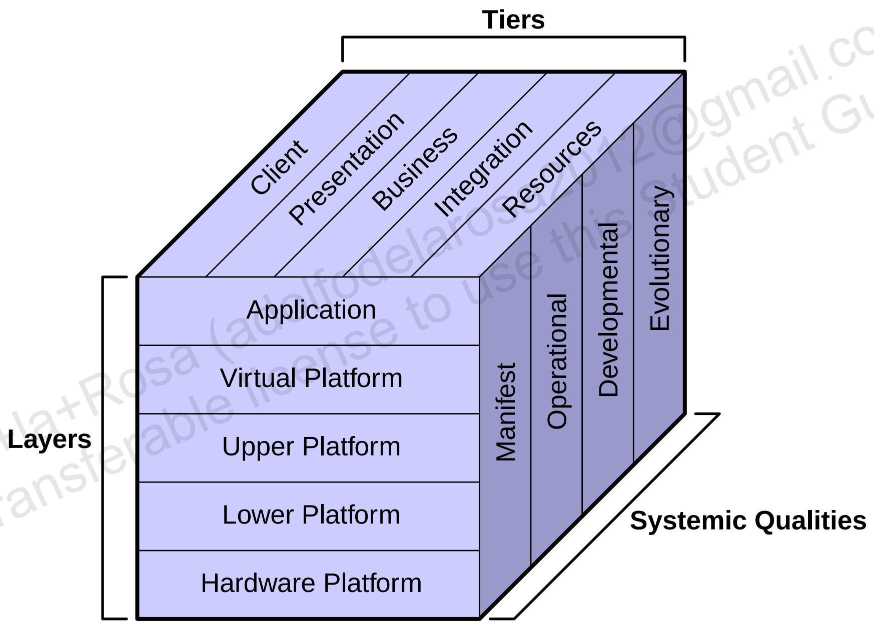

Figure 11-5 SunTone Architecture Methodology 3D Cube Diagram

Tiers

Tiers are "A logical or physical organization of components into an ordered chain of service providers and consumers." (SunTone Architecture Methodology page 10)

SunTone Architecture Methodology suggests the separation of the software system into five major tiers:

- Client – Consists of a thin client, such as a web browser.

The Client tier encompasses all of the components with which the user interacts. In a web application, these are usually the web pages and forms that the web server delivers to the web browser. This is called a thin client. In the Hotel Reservation System, the Manage a Reservation Online (use case number E5) use case will be implemented using a web-based thin client.

A fat client is a user interface that presents a rich user experience. For example, a spreadsheet is an application that requires a fat client due to the complexity of the user experience. In the Hotel Reservation System, the Manage a Reservation (use case number E1) use case will be implemented using a fat client.

- Presentation – Provides the web pages and forms that are sent to the web browser and processes the user's requests.

  The Presentation tier provides a buffer between a thin client and the Business tier. This tier accepts user actions in the form of an HTTP request, processes the request by calling the services of the Business tier, and responds to the HTTP request with the next view (a web page) for the user.

  A thick client might bypass the Presentation tier by accessing the Business tier directly.

- Business – Provides the business services and entities.

  The Business tier provides the components that manage the business logic, rules, and entities of the application. For example, services are usually components that manage the process of a use case, such as the `ResvService` component could be used to manage the creation and manipulation of reservations. Also, the `Reservation` component represents a business entity.

- Integration – Provides components to integrate the Business tier with the Resources tier.

  The Integration tier provides components that perform the fundamental CRUD (create, retrieve, update, and delete) operations for a given business entity.

- Resource – Contains all backend resources, such as a DataBase Management System (DBMS) or Enterprise Information System (EIS).

  The Resource tier includes all external systems that record business data.

Layers

Layers are "The hardware and software stack that hosts services within a given tier. (layers represent component to container relationships)" (SunTone Architecture Methodology page 11).

SunTone Architecture Methodology suggests the separation of the software system into five major layers:

- Application – Provides a concrete implementation of components to satisfy the functional requirements.

  The Application layer includes all of the components built by the development team to create the system solution. You can also purchase Application layer components from third-party vendors and integrate these with the custom-built components.
- Virtual Platform – Provides the APIs that application components implement.

  The Virtual Platform layer provides the specification of the component and container interfaces. For example, in a web application (on the Presentation tier) the Virtual Platform includes the specification of the servlet interface and the JSP language.
- Upper Platform – Consists of products such as web and EJB technology containers and middleware.

  The Upper Platform layer provides the set of packages that provide the infrastructure for the Application layer components. This includes the container used by the components. For example, in the Presentation tier the Upper Platform is usually a web container implementation such as Tomcat or Sun™ ONE Application Server.

  Java technology provides portability across almost all major Lower Platform layers because the Java™ Virtual Machine provides platform framework on top of the Lower Platform.
- Lower Platform – Consists of the operating system.

  The Lower Platform layer provides a description of the operating systems that support the previous layers. In particular, the Lower Platform layer is constrained by the platform specified by the container of the Upper Platform.
- Hardware Platform – Includes computing hardware such as servers, storage, and networking devices.

  The Hardware Platform layer provides a description of the hardware required to support the previous layers.

Systemic Qualities

"The strategies, tools, and practices that will deliver the requisite quality of service across the tiers and layers." (SunTone Architecture Methodology page 11)

SunTone Architecture Methodology suggests the categorization of systemic qualities into four main groups:

- Manifest – Addresses the qualities reflected in the end-user experience of the system.

  Examples of manifest qualities include: performance, reliability, availability, usability, accessibility, and so on.
- Operational – Addresses the qualities reflected in the execution of the system.

  Examples of operational qualities include: throughput, security, serviceability, and so on.
- Developmental – Addresses the qualities reflected in the planning, cost, and physical implementation of the system.

  Examples of developmental qualities include: realizability (the measure of confidence that a design can be easily realized in code) and planability (the measure of confidence on the predictability of determining cost estimates and planning to those estimates).
- Evolutionary – Addresses the qualities reflected in the long-term ownership of the system.

  Examples of evolutionary qualities include: scalability, extensibility, flexibility, and so on.

Architecture is a process of selecting trade-offs. Some systemic qualities will be more important than others. The architect must make the appropriate decisions that determine which systemic qualities to focus on.

Exploring the Architecture Workflow

The Architecture workflow analyzes the NFRs of the Requirements model to determine the structure of the infrastructure components in the proposed system. Infrastructure components are software components that support the structure of the design components in a way that satisfies the NFRs. For example, a performance or scalability NFR might require that business logic be placed on one or more distributed server hosts; by having multiple servers, the system can support more users at a specified level of performance. To make the business logic distributable, you need to develop the infrastructure to support this distribution, such as RMI or Common Object Request Broker Architecture (CORBA).

The Design workflow analyzes the FRs of the Requirements model to determine the structure of the design components. Design components are abstract software components[1] that support the behavior specified by one or more use cases. For example, design components include user interface elements, business services, and domain entities. A Design model captures the collaboration relationships between the design components to satisfy one or more use cases.

The Architecture model and the Design model are merged together to provide the first draft of a Solution model. It is the Solution model for which the software programmers can create code. Figure 11-6 shows this.

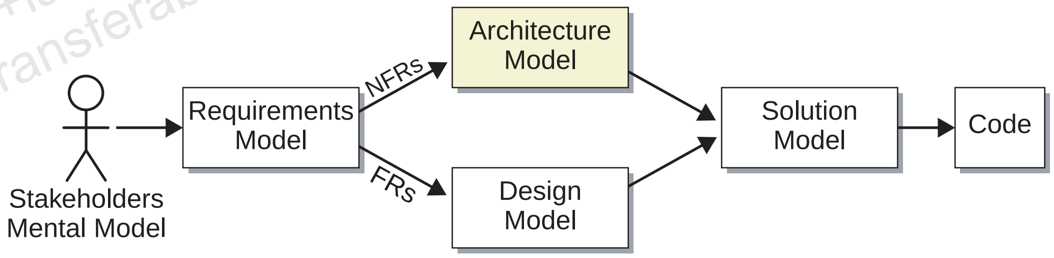

Figure 11-6 Architecture Model in the OOSD Process

The Architecture workflow is difficult to describe because the tasks involved vary greatly depending on which systemic qualities the architect focuses.

This course presents a simplified workflow:

1. Select an architecture type for the system.

1. Here "abstract" means that the design components do not map directly to code.

An architect selects the architecture type that best satisfies the high-level constraints and NFRs. An architecture type refers to a small set of abstract architectures, such as standalone, client/server, App-centric n-tier, web-centric n-tier, and enterprise n-tier. This step is discussed in “Selecting the Architecture Type” on page 11-38.

Based on the selected architecture type, a high-level Deployment diagram is created to show the distribution of the top-level system components on each hardware node.

2. Create a detailed Deployment diagram for the architecturally significant use cases.

   The architect creates a detailed Deployment diagram that shows the main components necessary to support the architecturally significant use cases. To create this diagram, the architect must create a Design model of the architecturally significant use cases and then place these abstract components into an infrastructure that supports the NFRs. Therefore, the architect must do some Design work to create the detailed Deployment diagram. This step is discussed in “Creating the Architecture Workflow Artifacts” on page 11-45.

3. Refine the Architecture model to satisfy the NFRs.

   The architect uses Architectural patterns to transform the high-level architecture type into a robust hardware topology that supports the NFRs. This workflow activity is beyond the scope of this course.

4. Create and test the Architecture baseline.

   The architect implements the architecturally significant use cases in an *evolutionary prototype*. When all architecturally significant use cases have been developed, the evolutionary prototype is called the *Architecture baseline*. The Architecture baseline represents the version of the system solution that manages all risks. The Architecture baseline is the final product of the Elaboration phase and becomes the starting point of the Construction phase.

   The Architecture baseline is tested to verify that the selected systemic qualities have been satisfied. The Architecture baseline is refined by applying additional patterns to satisfy the systemic qualities.

5. Document the technology choices in a tiers and layers Package diagram.

   For each tier and each layer, the architect identifies the appropriate technologies to be used in that tier and layer. This step is discussed in “Creating the Tiers and Layers Package Diagram” on page 11-49.

6. Create an Architecture template from the final, detailed Deployment diagram.

   The Architecture template is an abstract version of the detailed Deployment diagram in which the Design components are identified within the structure of the architecture (and infrastructure) components. You can use this template to guide the development team during the Construction phase.

   This is not a formal artifact of the Architecture workflow. However, you will use this template later in the course. This step is discussed in "Creating the Architecture Template" on page 11-47.

This workflow concludes with the creation of an Architecture model. For a complex system, this model can be large and complicated. In this course, the Architecture model includes the following:

- A high-level Deployment diagram

  This diagram shows the distribution of the top-level system components on each hardware node. This diagram is used to show how the top-level software components will be deployed.

- A detailed Deployment diagram

  This diagram shows the detailed components that satisfy one or more use cases and how those components are organized within the high-level structure of the system.

- An Architecture template

  This template shows the essential structure of the detailed Deployment diagram, but without referencing specific Design components. A designer can use this template to plug in the Design components into the Architecture model. This merging of the Design model with the Architecture model results in a Solution model, which guides the construction of the physical components.

  The Architecture template is not a formal artifact, but in this course you will use it to determine how to create the Solution model from the Design model.

- A tiers and layers Package diagram

  This diagram shows a matrix of packages in which the technologies selected by the architect are recorded.

- An Architecture baseline

  This artifact is an actual working system that implements all of the architecturally significant use cases. This code base is the starting point of the development team during the Construction phase.

Design Model

During the Architecture workflow, the architect performs the job of the software designer by creating Design models of the architecturally significant use cases. Remember not all models require artifacts; therefore, the architect might not create a drawing of the Design model. Nevertheless, the architect definitely has a mental model of the design of the use cases. Figure 11-7 illustrates an example design model.

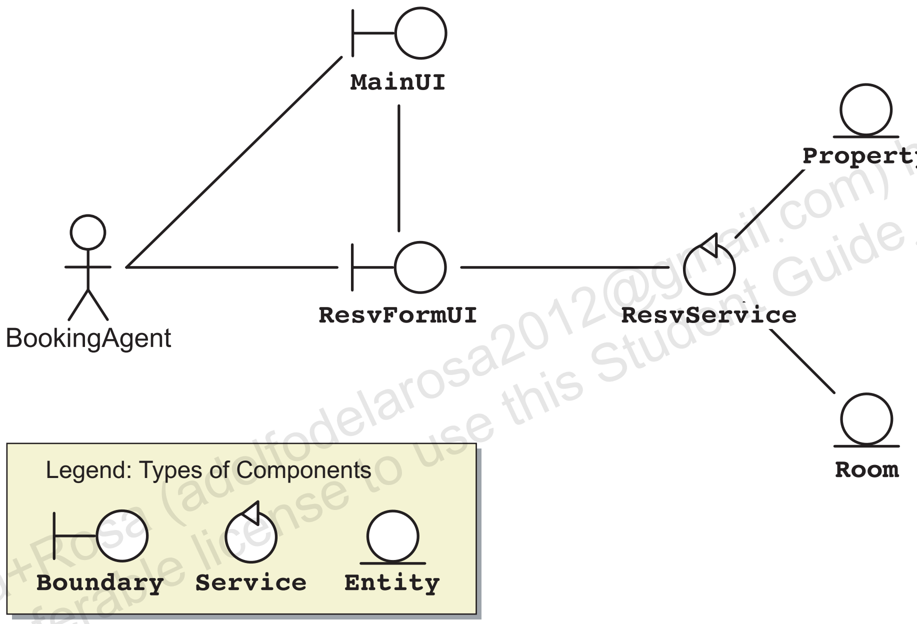

Figure 11-7 Example Design Components

Module 8, "Transitioning from Analysis to Design Using Interaction Diagrams," describes how to create these models. This module only describes the three main types of designed components:

- Boundary

  Boundary components are user interface components that communicate with a human actor. Boundary components can also represent adaptors to external software systems.

- Service

  Service components perform business operations. Some Service components can be designed to manage the workflow of a use case.

- Entity

  Entity components represent the domain objects of the system.

Architecture Template

An Architect creates an architecture template. This template provides a view of the set of physical components that implement the Design components. The template consists of the types of components that will implement each type of Design component: boundary, service, and entity. The template also shows the communication dependencies (shown by dashed arrows). Figure 11-8 illustrates an example architecture template.

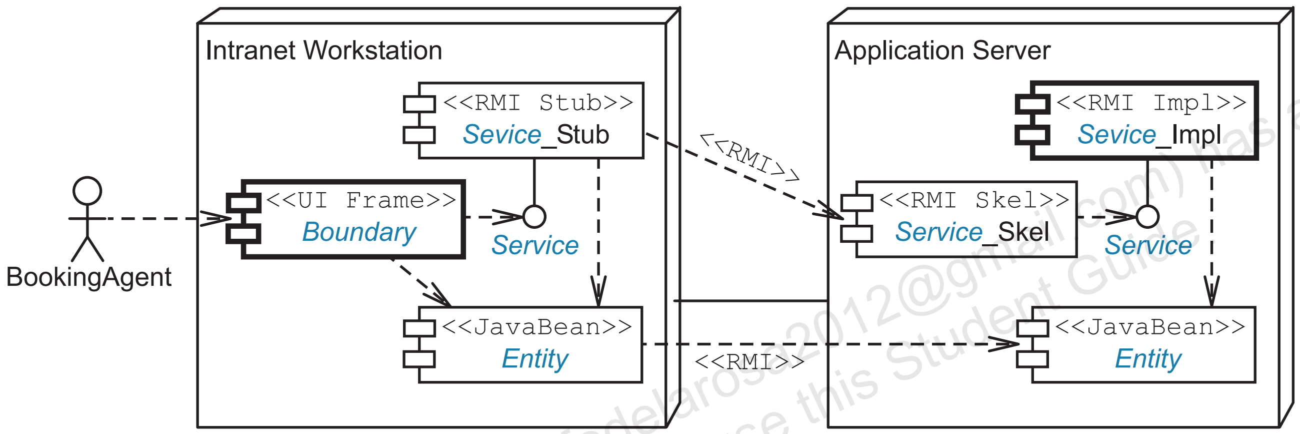

Figure 11-8 Example Architecture Template

The next two modules describe how to create these architecture templates. For now, this module only describes the relationship between the Architectural components and the Design components. The Design components implement use case functionality. The Architectural components describe the structure of the Design components that satisfy the quality of service goals.

Solution Model

A Solution model is formed by *snapping in* the Design components into the Architecture template. This model forms the basis of the actual implementation. Figure 11-9 illustrates an example Solution model.

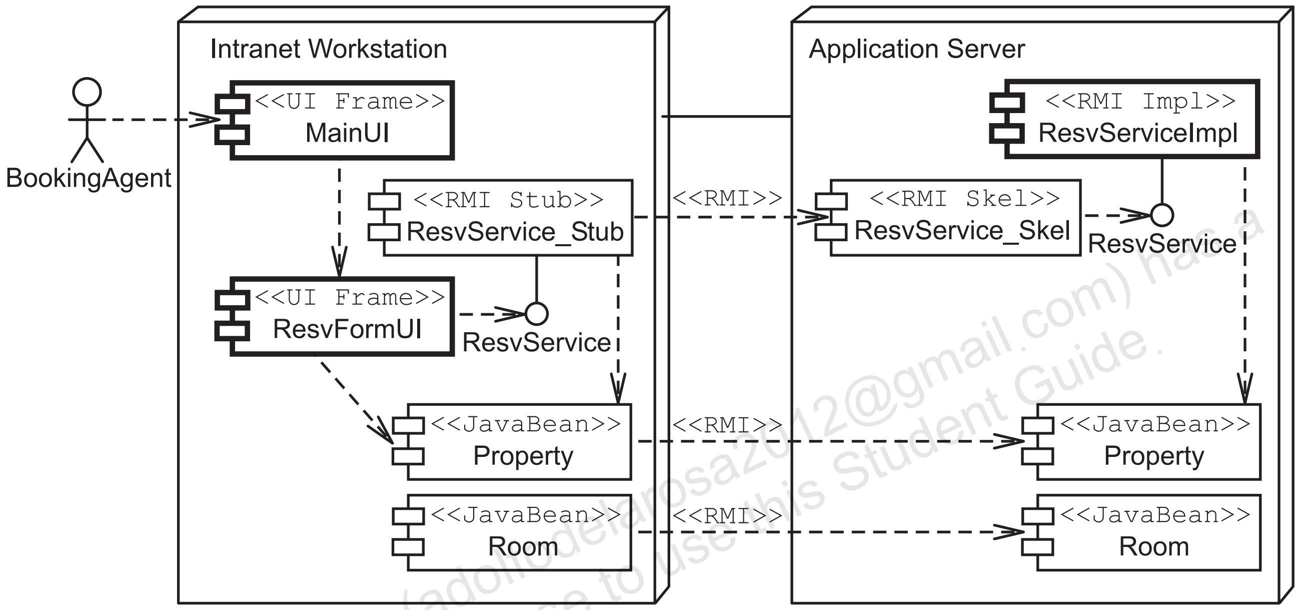

Figure 11-9 Example Solution Model

Architectural Views

The views of the Architecture model take many forms. Some elements (such as risk mitigation plans) are documented with text. Others can be recorded using the following UML diagrams:

- Package diagrams
- Component diagrams
- Deployment diagrams

These diagrams are discussed in the next three sections.

Describing Key Architecture Diagrams

The views of the Architecture model use three types of UML diagrams: Package, Component, and Deployment.

Identifying the Elements of a Package Diagram

A package is "A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages." (UML v1.4, page B-14)

A UML Package diagram shows dependencies between packages. A package is a UML icon that groups other kinds of UML element, diagram, or additional packages. Figure 11-10 shows two graphical forms for a package.

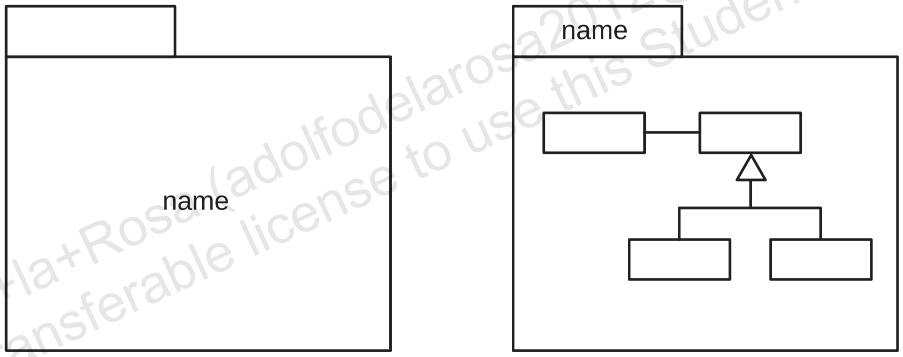

Figure 11-10 Elements of a UML Package Diagram

The package name can be placed in the body box (left icon in Figure 11-10) or in the name box (right icon in Figure 11-10). Use the body box icon when you do not want to show the contents of the package. Use the name box icon when you do want to show the contents. Use this form of the package icon to represent the package abstractly (when you do not want to show the contents).

Note – A UML package is *not* the same as a Java technology package. However, you can use a UML package to represent a Java technology package.

For example, you can group your UI components into a UI package, your services into the Services package, and your entities into a Domain package. The UI components depend on the business services and domain entities, and the service components depend on the domain entities. Figure 11-11 shows these dependencies.

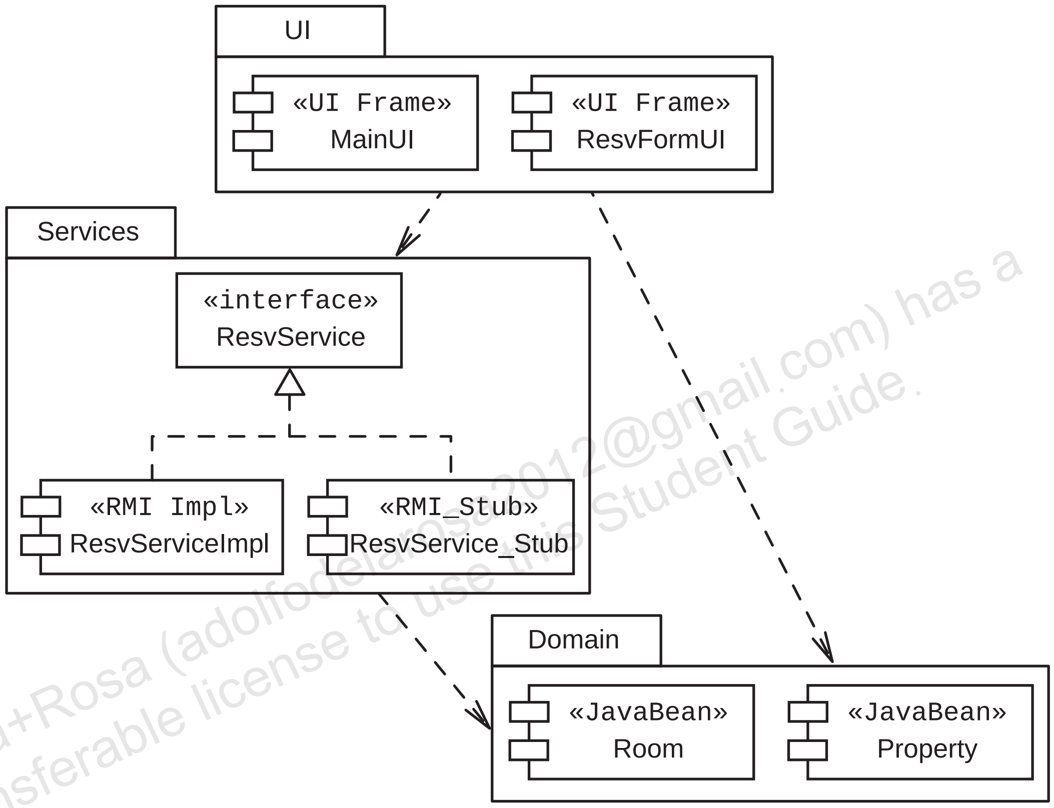

Figure 11-11 An Example Package Diagram

Figure 11-12 shows an alternate, more abstract, version of this Package diagram.

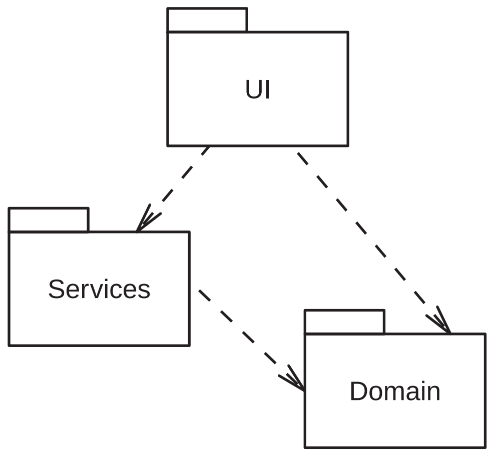

Figure 11-12 An Abstract Package Diagram

Identifying the Elements of a Component Diagram

A Component diagram is "A diagram that shows the organizations and dependencies among components." (UML v1.4 page B-14)

Purpose of a Component Diagram

A *Component diagram* represents physical, software pieces of a system and their relationships. When one component (or object) collaborates with another component, this collaboration is illustrated with a dependency between the *client* component and the *service* component. Figure 11-13 shows an example Component diagram.

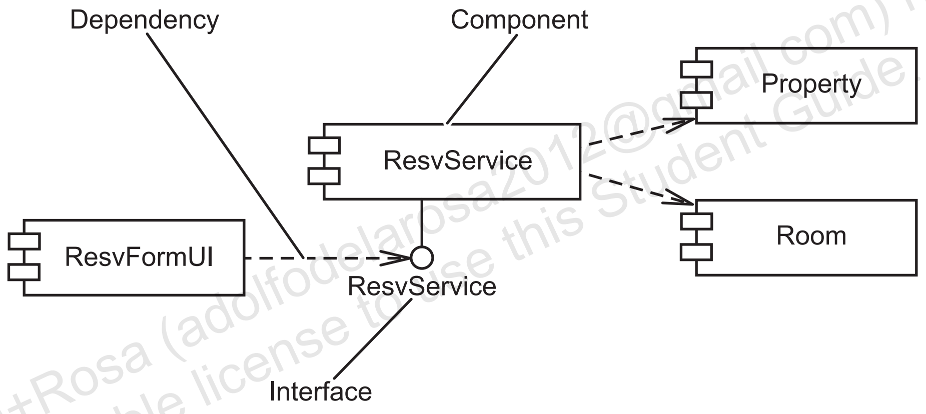

Figure 11-13 An Example Component Diagram

Figure 11-14 shows the notation for a required interface in UML 2, which is particularly useful when the component with the required interface is shown in a separate diagram.

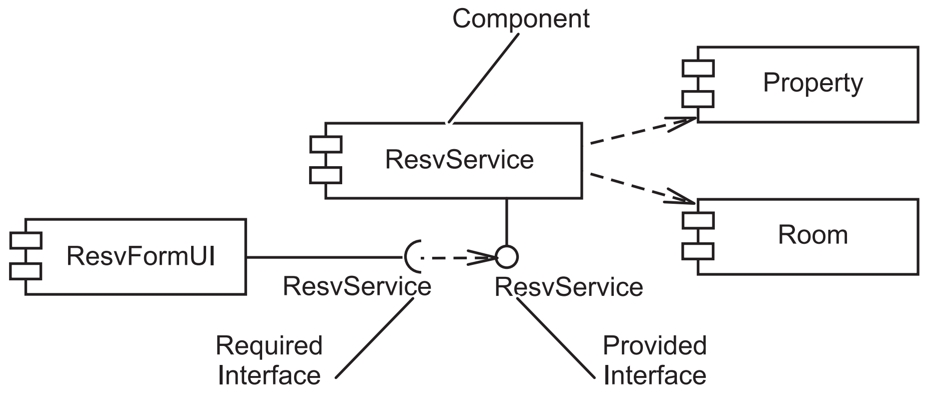

Figure 11-14 An Example of the UML 2 Style Required Interface

Figure 11-15 shows a UML 2 style Component notation that may be used instead of the previous style.

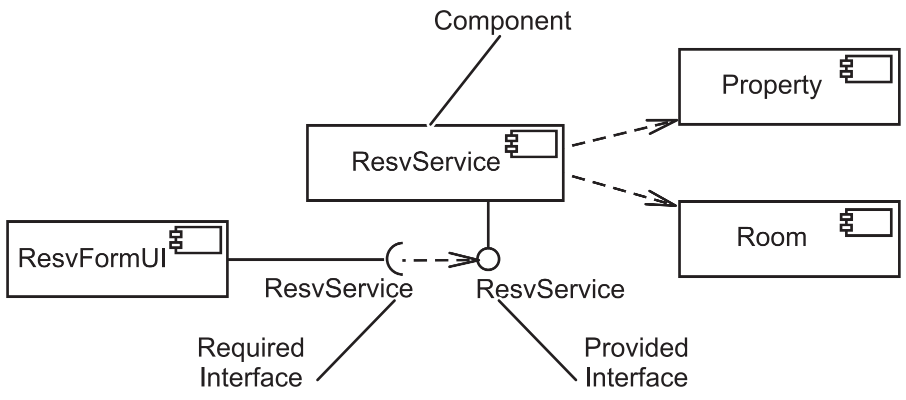

Figure 11-15 An Example of a UML 2 style Component Notation

Characteristics of a Component

You can use a UML Component diagram in a wide variety of ways. A few characteristics of components and Component diagrams are:

- A component represents any *software unit*.

  A component can be any type of software, including non-executable software. Examples include JavaBean components, source code modules, HTML files, and so on.
- A component can be large and abstract.

  A component can be a complete system or subsystem. For example, a DBMS server can be considered a single component.
- A component can be small.

  A component can also be a single object; therefore, a class can represent a set of component objects.
- A component might have an interface that it exports as a service to other components.

  An interface can be conceptual or concrete. An abstract interface might be the communication protocol used by a large component; for example, structured query language can be considered the interface to communicate to a DBMS. A concrete interface describes the set of methods implemented by some component. In Java technology, an interface declaration serves this purpose.
- A component can be a file, such as a source code file, an *object* file, a complete executable, a data file, HTML or media files, and so on.

  Many things can be components. Not all of them will be executable. It is perfectly valid to think of an HTML file as a static component. Components can also be collections of other components. For example, a `.jar` file is a collection of class components.

Types of Components

As mentioned above, many types of things can be components. The top two icons in Figure 11-16 show a generic component; the component on right side uses a line with a circle to show that this component implements an interface. The icons on the bottom row of Figure 11-16 show several different types of components and their icons.

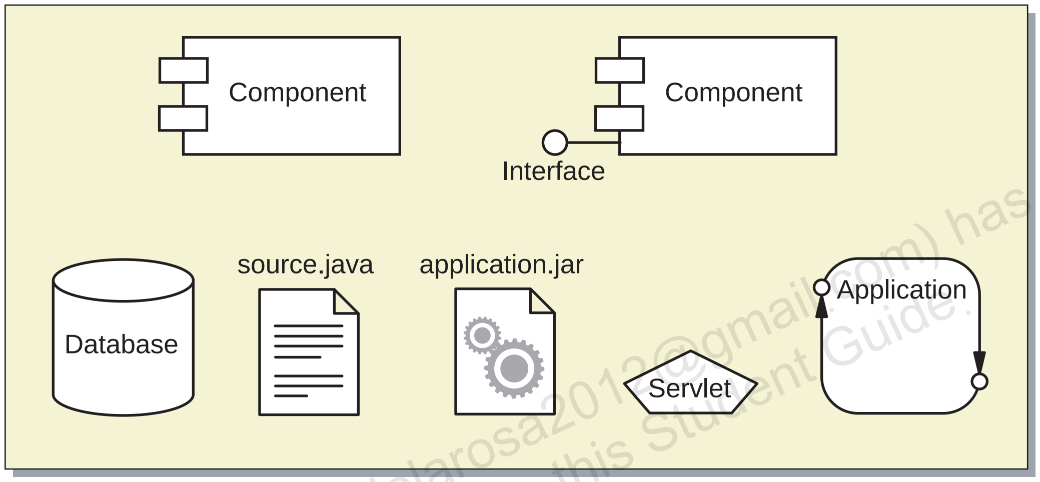

Figure 11-16 Types of Software Components

UML permits special icons to represent specific types of components, such as a database, a text file, or an executable file. You can create icons that are meaningful to your project. For example, in this course a Java servlet component is represented by a pentagon and an application component is represented by a rounded-corner rectangle with the arrow-circle icons on the sides.

Example Component Diagrams

The most common use for Component diagrams is to show the dependencies (or collaborations) between various software components. Figure 11-17 shows an example of such a Component diagram.

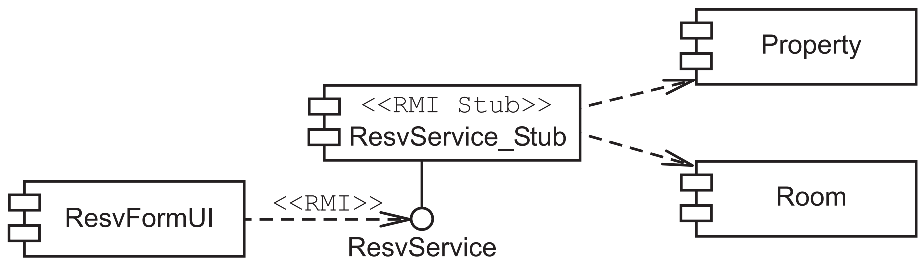

Figure 11-17 Component Diagrams Can Show Software Dependencies

Figure 11-18 shows an example of such a Component diagram with a Web Service component dependency.

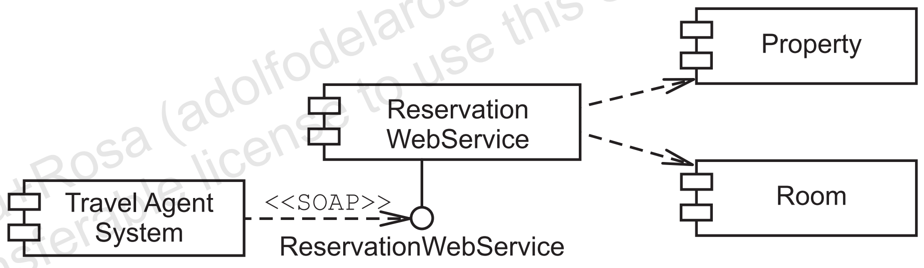

Figure 11-18 Component Diagram Showing a Web Service Dependency

Another common use is to show the organization of software components. For example, you can show how the `domain.jar` file is built from the class files for the `Customer` and `Reservation` classes; and how these class files are built from the source files. Figure 11-19 shows an example of such a Component diagram.

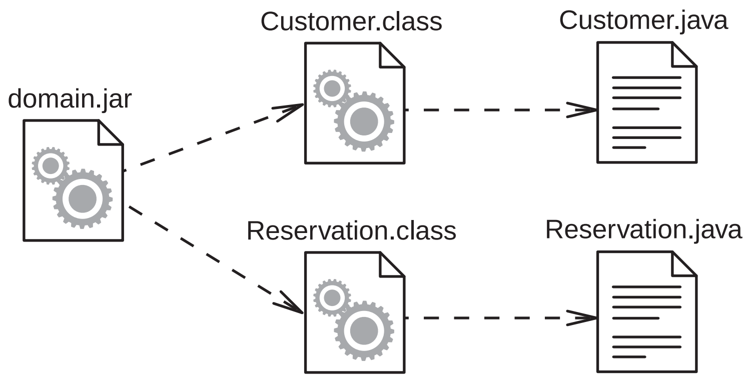

Figure 11-19 Component Diagrams Can Represent Build Structures

The Component diagram is one of the most flexible diagrams in UML. You can create new views of your systems by experimenting with different uses of Component diagrams. Another useful view that uses Component diagrams is the Deployment diagram.

Identifying the Elements of a Deployment Diagram

A Deployment diagram is "A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them." (UML v1.4, page B-7)

A *Deployment diagram* represents physical, hardware pieces of a system, the distribution of software components within each hardware node, and the dependencies between the software components. Figure 11-20 illustrates an example Deployment diagram.

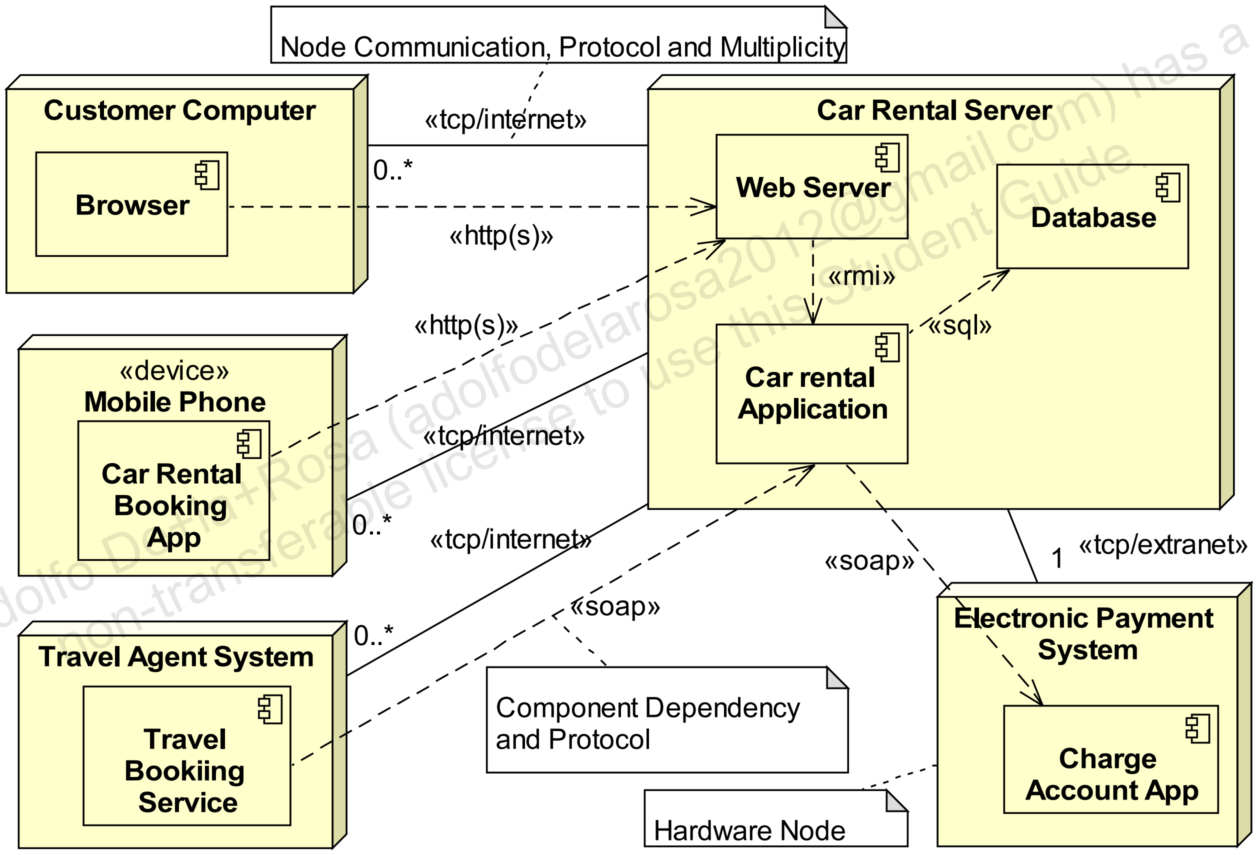

Figure 11-20 An Example Deployment Diagram

Purpose of a Deployment Diagram

A Deployment diagrams contain the following elements:

- Hardware nodes can represent any type of physical hardware.

  A hardware node is usually a computer, but it can also be any physical device that communicates with a computer network. Such devices include printers, scanners, personal digital assistant (PDA), cell phone, network routers, firewalls, and so on.

- Links between hardware nodes indicate connectivity and can include the communication protocol used between the nodes.

  A Deployment diagram represents a graph of links that show the topology of a real computer network. These links are shown as solid lines with no arrows. There is usually a stereotype label on the link, that specifies the communication protocol used between the nodes.

- Software components are placed within hardware nodes to show the distribution of the software across the network.

  A Deployment diagram can show how to deploy software components onto the hardware nodes of the system solution. These software components are usually large-scale components, such as complete executable applications and class libraries.

Types of Deployment Diagrams

Deployment diagrams come in two forms:

- A *descriptor* Deployment diagram shows the fundamental hardware configuration.

  This is a conceptual diagram. The hardware nodes in this form are meant to be thought of as typical pieces of hardware. Figure 11-20 on page 11-35 is an example of a descriptor Deployment diagram.
- An *instance* Deployment diagram shows the hardware configuration and includes specific hardware decisions.

  This form represents an actual or planned hardware topology. The hardware nodes in this type of Deployment diagram uses notation similar to Object nodes in which the name text is underlined and the full name includes the name of the node and the type of hardware for that node. Figure 11-21 illustrates an example of this.

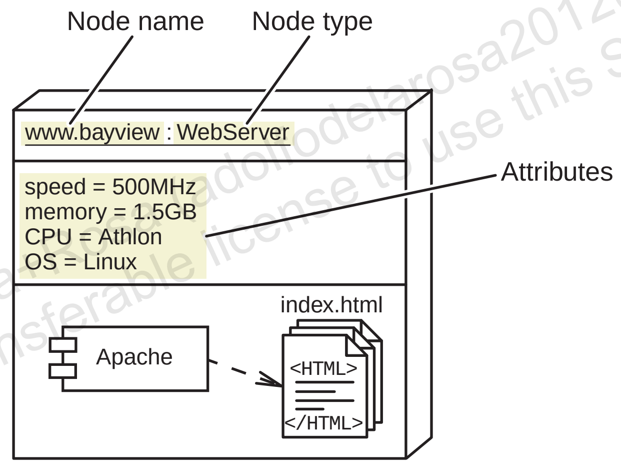

Figure 11-21 An Example Instance Hardware Node

Notice that you can specify attributes of a hardware node. Example attributes include the speed and memory capacity of the computer.

Selecting the Architecture Type

Selecting the architecture type for a system provides the development team a vision of the top-level software and hardware structure of the system. This is recorded in a high-level Deployment diagram.

The architecture to use depends on many factors, including:

- The platform constraints in the system requirements

  Very often the SRS will specify the platform (for example, operating system, application platform, and hardware) to use when deploying the system. This might constrain the architecture type.
- The modes of user interaction

  How a given actor will use the system might constrain the architecture. For example, the Hotel Reservation System has a requirement to provide a web user interface for customers. This requirement will guide several architectural decisions; for example, the system must include a web server.

  There are many modes of interaction; some examples are GUI, Web, voice, direct manipulation, handheld devices, and so on.
- The persistence mechanism

  The type of persistent storage will guide architecture decisions. These issues are described in Module 12, “Introducing the Architectural Tiers”.
- Data and transactional integrity

  Similarly, data and transactional integrity will guide architecture decisions. These issues are beyond the scope of this course.

There are hundreds of successful software architectures. A few common types are:

- Standalone applications
- Client/Server (2-tier) applications
- N-tier applications
- Web-centric n-tier applications
- Enterprise n-tier applications

Using the selected architecture type, the architect provides an essential artifact of the Architecture workflow: the high-level Deployment diagram. This artifact shows the fundamental hardware topology as well as the top-level components that are hosted by each hardware node. These top-level components are usually independent applications.

Standalone Applications

Standalone applications exist as a single application component hosted by the user's workstation. Figure 11-22 illustrates this architecture type.

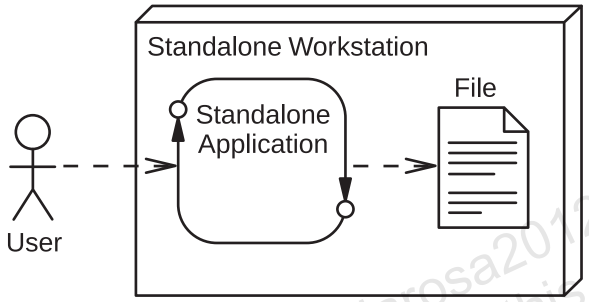

Figure 11-22 Generic Standalone Architecture Type

Standalone applications are very common. Examples include word processors, graphics applications, text editors, and so on. These types of applications to not access a central data store (such as a DBMS) and do not participate in network communication.

This architecture type requires updating every user's workstation when the application changes. This is called the *deployment problem*. For most standalone applications this is not a serious problem. The user decides when to upgrade. It is only a problem when the user must share application-generated files with other users. Different versions of the software might have incompatible file formats.

Client/Server (2-Tier) Applications

Client/server applications are client applications hosted by the user's workstation that communicate to a common data store. Figure 11-23 illustrates this architecture type.

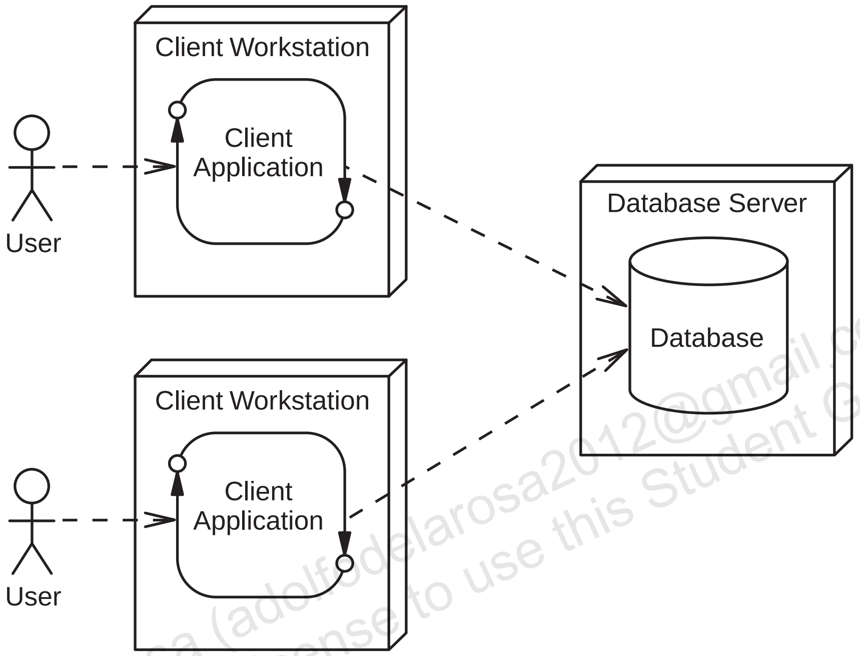

Figure 11-23 Generic Client/Server Architecture Type

There are two subtypes of client/server applications: thin client and thick client. A thin client refers to a UI that does not contain any business logic. In a client/server application a thin client is usually a forms-based front-end to a database. A thick client refers to a UI that provides business logic on the client tier. Thick clients can provide a more rich user experience. In the first case, the data store manages the data and transactional integrity rules. In the second case, the client application provides this management.

This architecture type also suffers from the deployment problem. This problem is significant in this situation, because every client must have the latest version of the client application when the data store is upgraded. Every user must be using the same version of the software or data corruption problems might occur.

N-Tier Applications

N-tier applications exist across multiple machines. There are three fundamental variations of n-tier applications:

- Application-centric
- Web-centric
- Enterprise

Application-centric applications partitions the business components onto a separate application server. The client workstations host a thin-client application that uses the services of the application server. Figure 11-24 illustrates this architecture type.

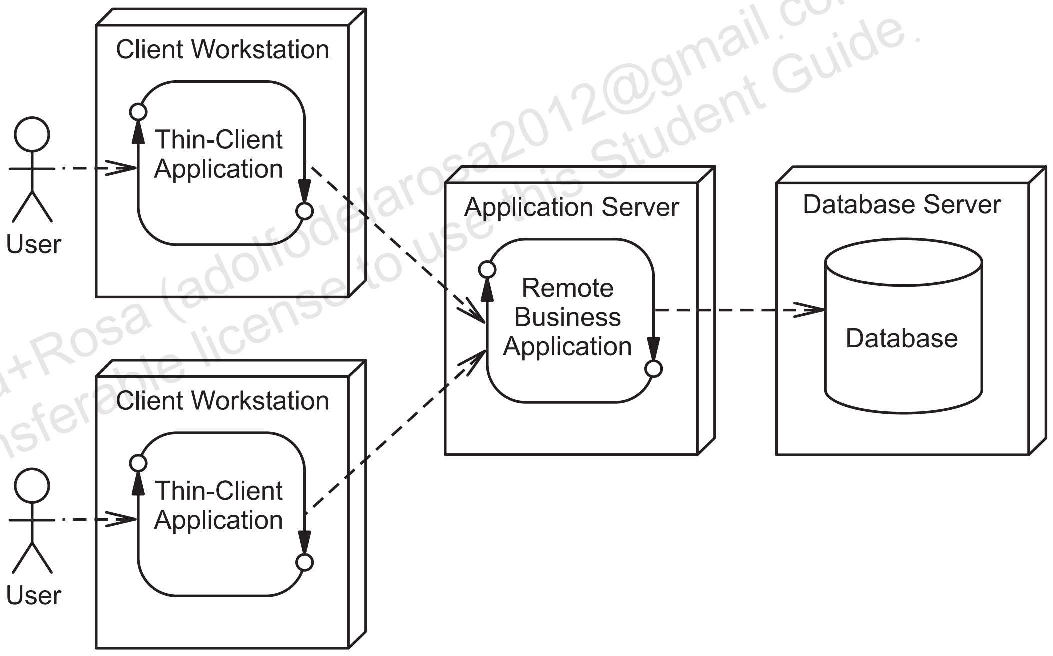

Figure 11-24 Generic N-tier Architecture Type

In this configuration, the application server manages the data integrity and performs all data store operations.

By separating the client applications from the business service application, these two high-level components can up upgraded independently. For example, if a bug in the business application is fixed, then only the application server needs to be upgraded.

Web-centric applications provide access to the application through a Web browser on the client workstation. You can use this architecture type to bring your business to the Internet. Figure 11-25 illustrates this architecture type.

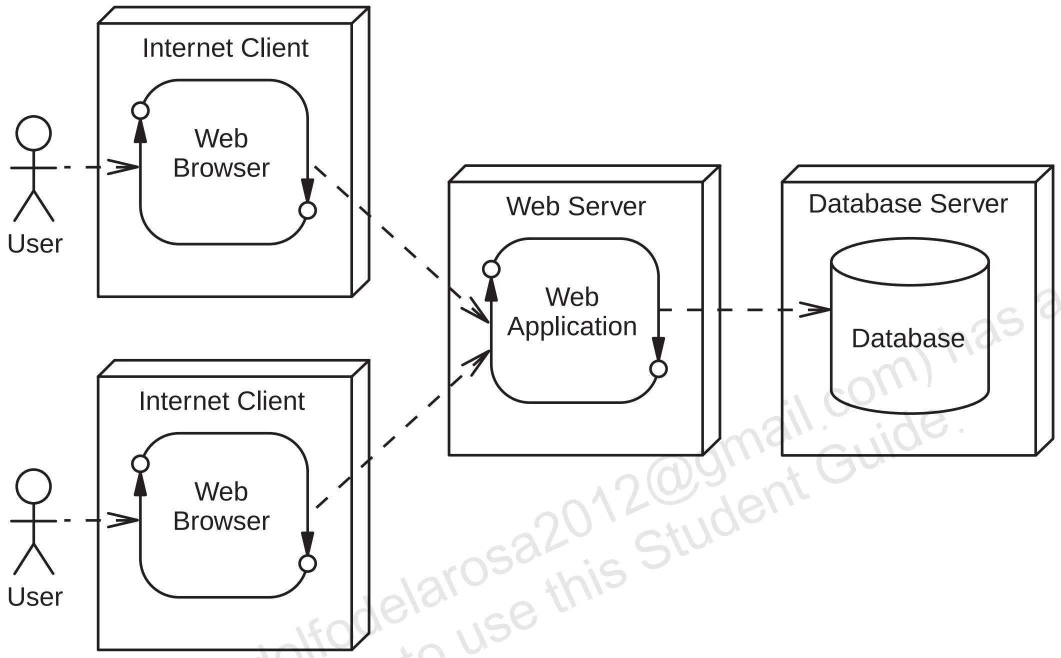

Figure 11-25 Web-Centric N-Tier Architecture Type

There is one major drawback to this architecture. If users within your company need to use the application, they must use the Web application. This is a security risk because you are making internal business functions accessible to the Internet. Alternatively, there could be thick-client applications that access the database directly in a client/server architecture. This could cause data integrity problems if the business logic in the Web application is different than the business logic in the intranet client applications.

The deployment problem is virtually invisible to the end users. After the Web application is upgraded, all users will immediately see the new application.

Enterprise applications provide a hybrid of web-centric and application-centric architectures. In this architecture type a Web application is provided for the Internet public, as well as thin-client applications for the company's intranet users. Figure 11-26 illustrates this architecture type.

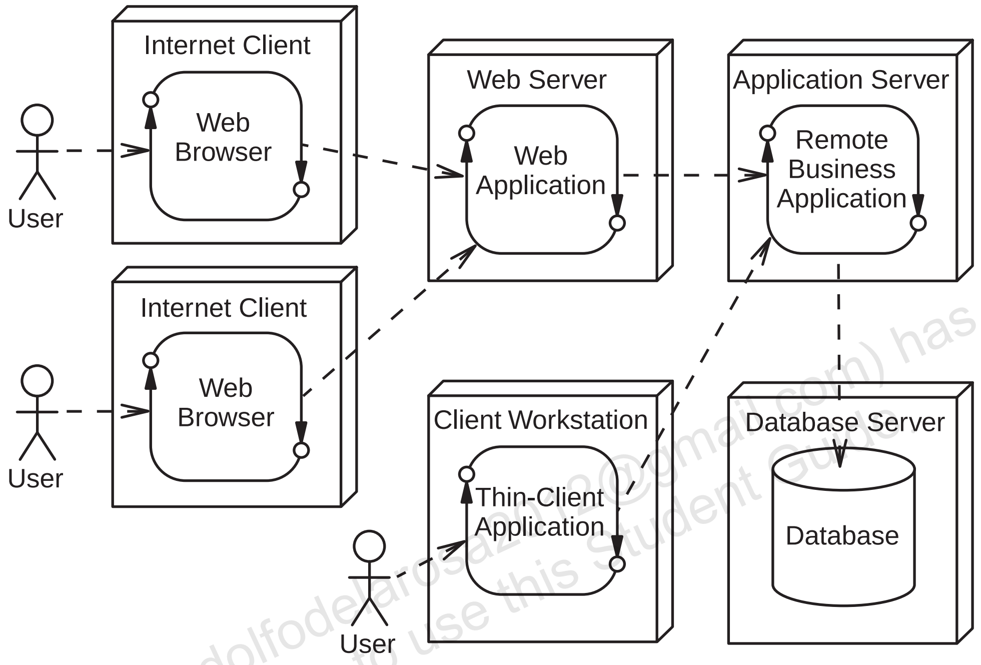

Figure 11-26 Enterprise N-Tier Architecture Type

This architecture type provides the greatest flexibility by separating each logical tier onto a separate machine. The data integrity and security problems with the web-centric architecture are resolved by having access to remote business services.

The main drawback of this architecture type is the complexity of managing all of the disparate software components on multiple workstations and servers.

Hotel Reservation System Architecture

For the Bay View case study, the ACME Consultants have selected the Enterprise n-tier architecture type because the system requires both a Web application for customers and an internal business application. Figure 11-27 illustrates this high-level architecture.

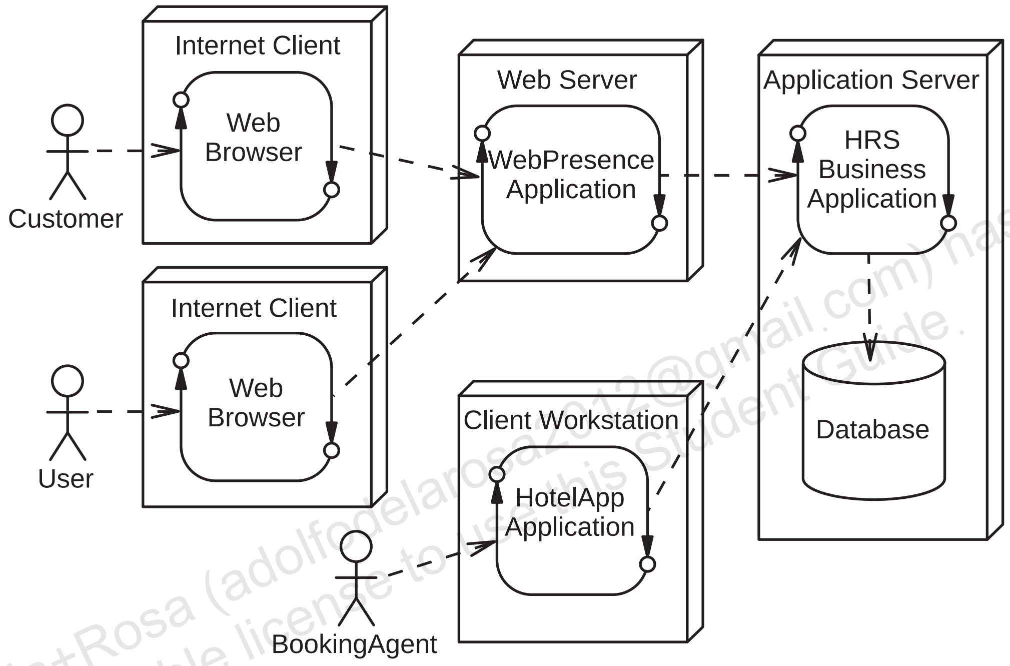

Figure 11-27 High-Level Deployment Diagram of the Hotel Reservation System

Creating the Architecture Workflow Artifacts

This section discusses how to create three key Architecture artifacts: the detailed Deployment diagram, the Architecture template, and the tiers and layers Package diagram.

Creating the Detailed Deployment Diagram

This section describes how to build a Detailed Deployment diagram. This is an essential part of the Architecture model because it shows the actual components that need to be developed. The Detailed Deployment diagram also acts as a basis for the Architecture template which the development team will use to complete the construction of the system after the architecture team has built the Architecture baseline.

To create a Detailed Deployment diagram:

1. Design the components for the architecturally significant use cases.

   This step requires the architect to design the boundary, service, and entity components that support each architecturally significant use cases.

2. Place Design components into the Architecture model.

   The Design components are then placed within an infrastructure that supports the high-level architecture. Therefore, if there is a Presentation tier, the servlet and JSP page components are required.

3. Draw the detailed Deployment diagram from the merger of the Design and Infrastructure components.

   The merger of the Design components with the infrastructure is then drawn using a detailed Deployment diagram.

Figure 11-28 on page 11-46 shows an example detailed Deployment diagram for the Hotel Reservation System. Two architecturally significant use cases were identified for this system: E1 “Manage a Reservation” and E5 “Manage a Reservation Online.” These two use cases have different UIs, a Web interface for customers on the Internet and a GUI interface for employees of the Bay View company. Both of these UIs communicate to a common application server using RMI.

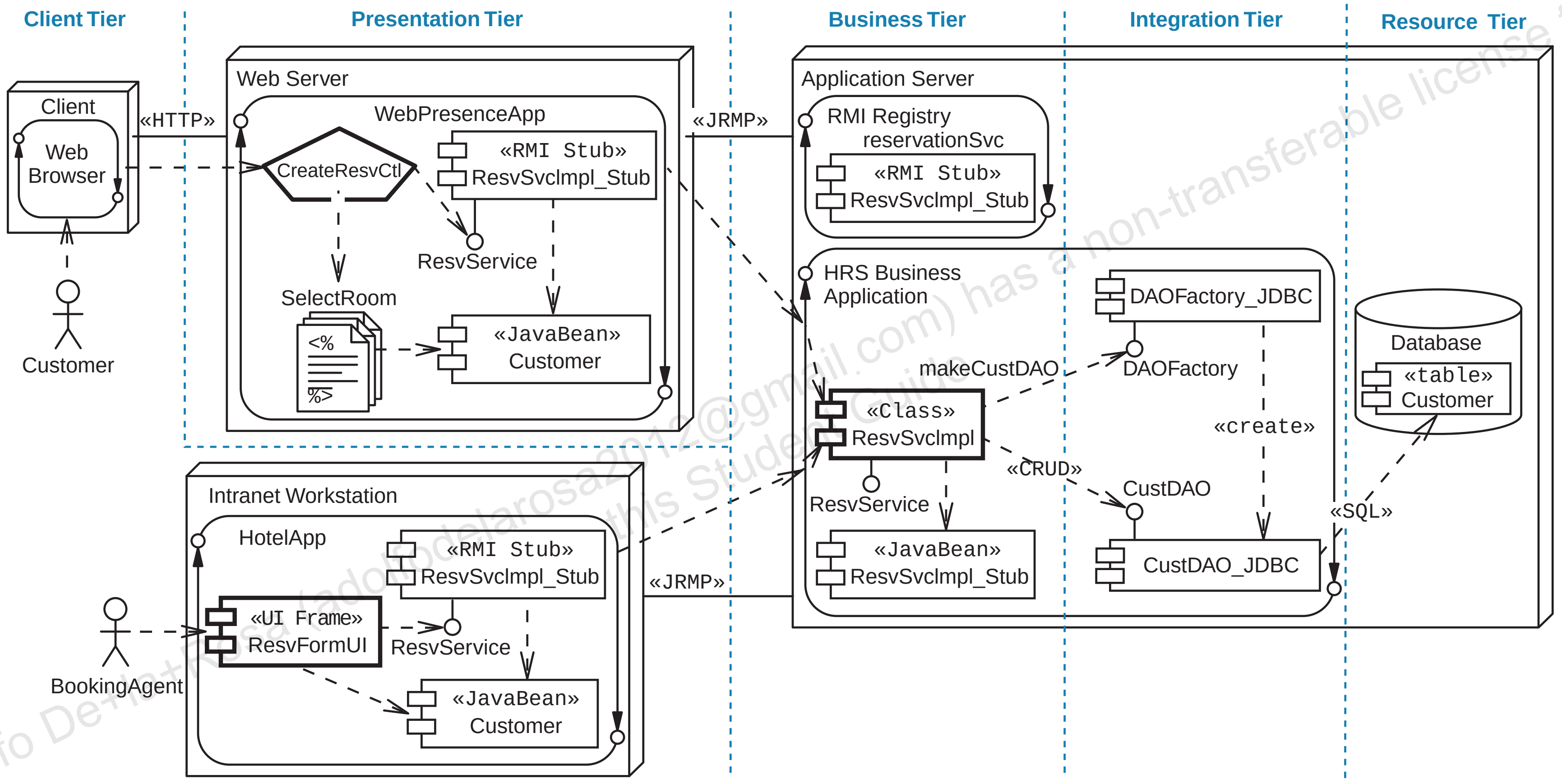

Figure 11-28 Example Detailed Deployment Diagram

Creating the Architecture Template

To create an Architecture template:

1. Strip the detailed Deployment diagram to just one set of Design components: boundary, service, and entity.

   The idea is to show the types of infrastructure components needed to support the Design components. You can simplify the detailed Deployment diagram by eliminating all redundant components. For example, only include one Entity type component (in each tier).

2. Replace the name of the Design component with the type (for example, `ResvSvcImpl_Stub` becomes *`Service`*`Impl_Stub`).

   This step provides a simplified map between the infrastructure components and the Design components by removing the specific names with the generic type names: Boundary, Service, and Entity.

Figure 11-29 on page 11-48 shows an example Architecture template.

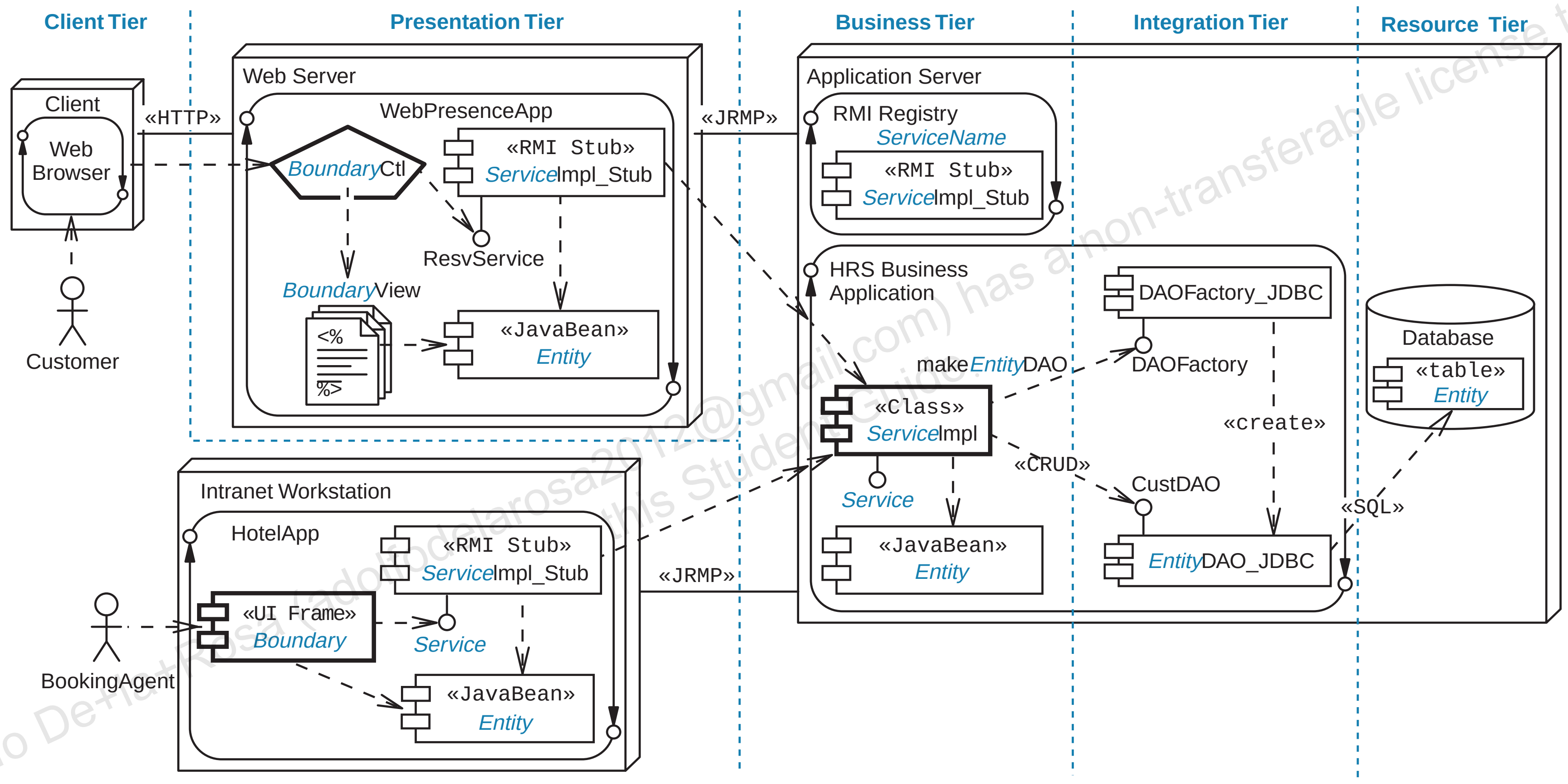

Figure 11-29 Example Architecture Template

Creating the Tiers and Layers Package Diagram

The tiers and layers Package diagram provides a matrix of package icons that list the technologies required for each combination of tier and layer. You can create this diagram by following these steps:

1. Determine what application components exist.

   In the Application layer, document the top-level components (usually applications) that exist for that tier. For example, the Presentation tier package would contain the WebPresenceApp component.

2. Determine what technology APIs, communication protocols, or specifications that the components require.

   In the Virtual Platform layer, document each technology required to build the components at the Application layer. For example, the Presentation tier package includes the Java servlet and JSP technology specifications. You should also include the version number of the specification.

3. Determine which container products to use.

   In the Upper Platform layer, document the products that act as containers to the Application layer components. For example, the Presentation tier package includes Tomcat and Java™ 2 Standard Edition (J2SE™) platform.

4. Determine which operating system to use.

   In the Lower Platform layer, document the operating system that will execute that the given tier's components.

5. Determine what hardware to use.

   In the Hardware Platform layer, document the type of machine that will host the given tier's components. Be as specific as you can with details about the number of processors, processor speed, memory capacity, and storage capacity.

Figure 11-30 shows an example tiers or layers Package diagram for the HotelApp application. The HotelApp is a thin 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 UI.

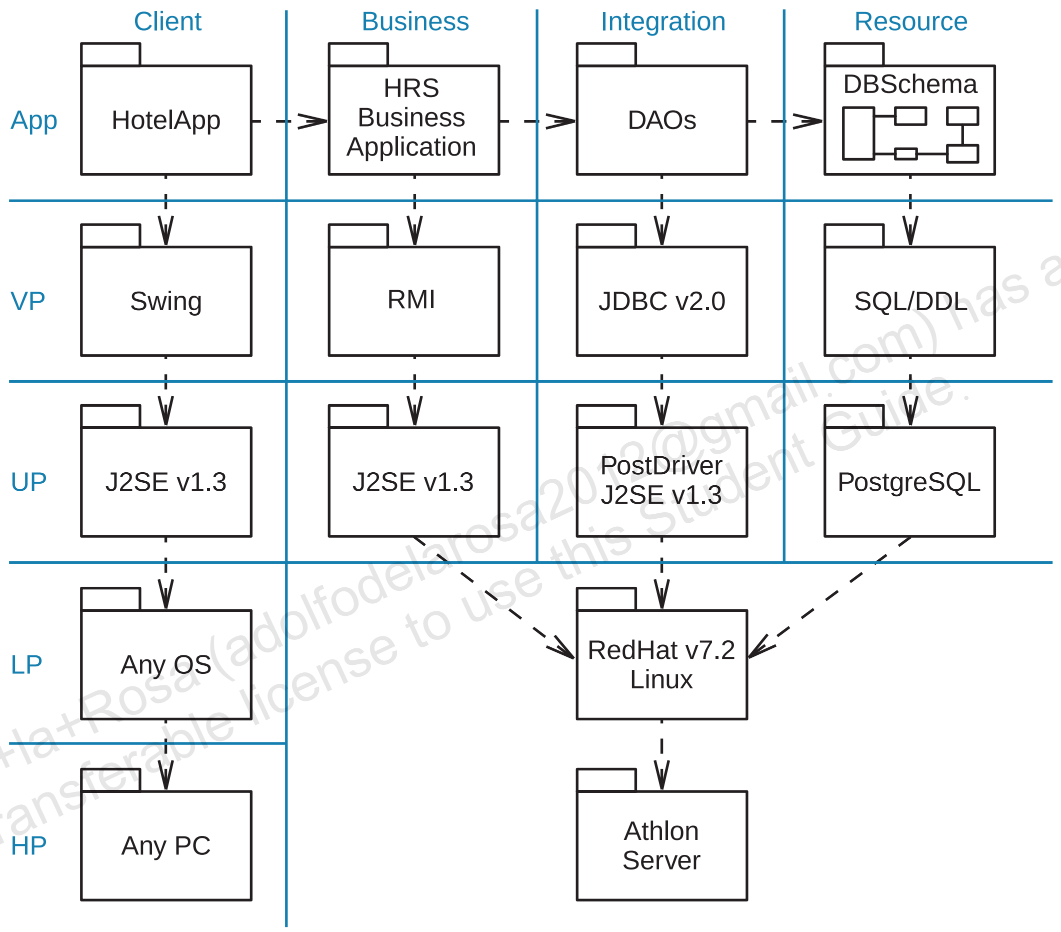

Figure 11-30 Tiers and Layers Diagram for the HotelApp

Figure 11-31 shows the tiers and layers Package diagram for the Hotel System's Web Presence. This is a Web boundary of the Hotel System application, so the Presentation tier is included to record the technologies used on the web server.

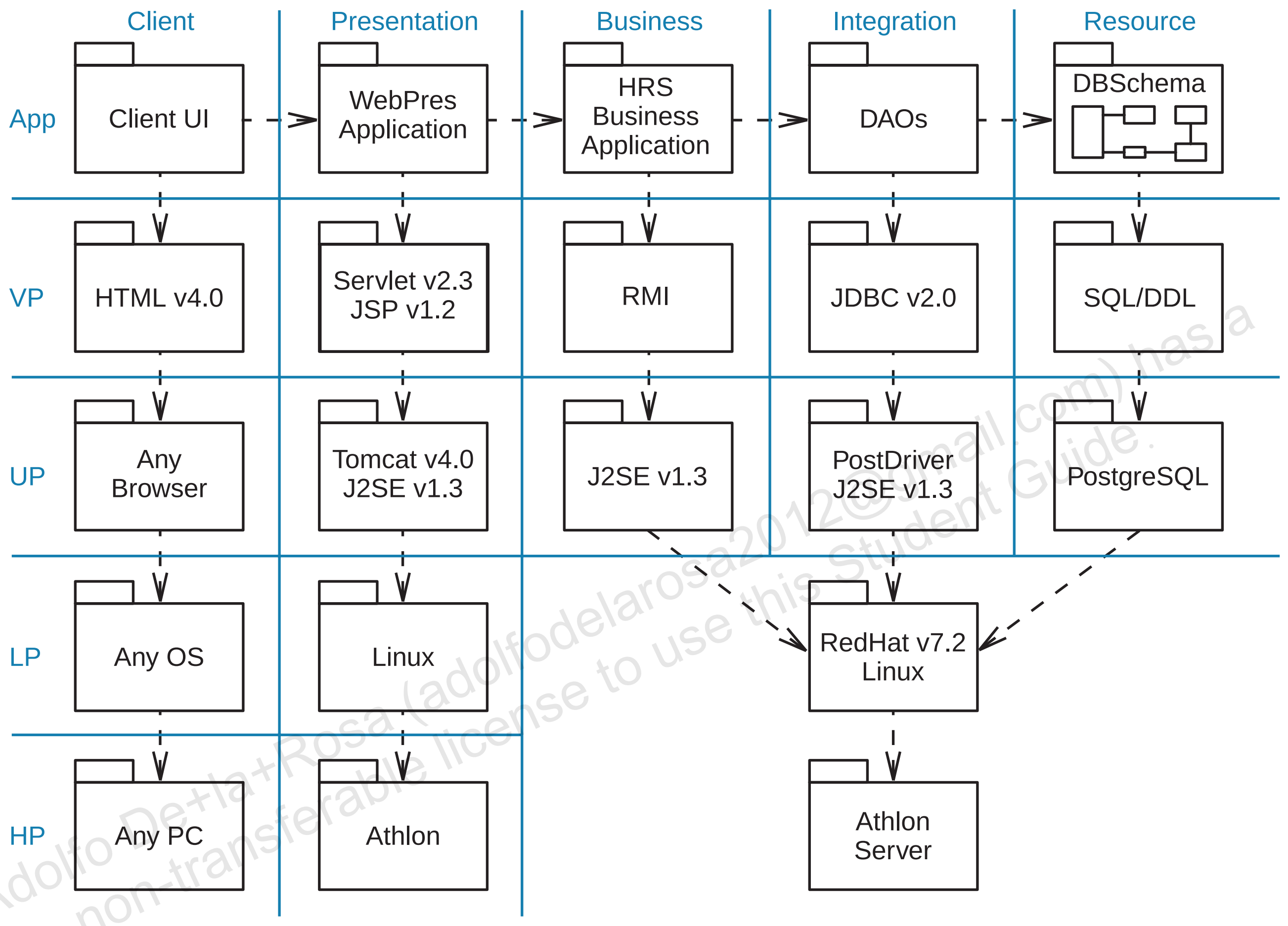

Figure 11-31 Tiers and Layers Diagram for the Hotel System's Web Presence

Summary

In this module, you were introduced to the key architectural concepts and diagrams. Here are a few important points:

- Difference between architecture and design:
  - Design produces components to implement a use case.
  - Architecture provides a template into which the designed components are realized.
- You can model the architecture using:
  - Deployment diagrams
  - Component diagrams
  - Packages
  - Tiers and layers