

# **Object-Oriented Analysis and Design Using UML**

**Volume I • Student Guide**

**OO-226 Rev E**

D61808GC21

Edition 2.1

June 2010

D67900

**ORACLE®**

**Copyright © 2010, Oracle and/or its affiliates. All rights reserved.**

#### **Disclaimer**

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

#### **Sun Microsystems, Inc. Disclaimer**

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

#### **Restricted Rights Notice**

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

##### **U.S. GOVERNMENT RIGHTS**

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

#### **Trademark Notice**

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This page intentionally left blank.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

This page intentionally left blank.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Table of Contents

---

<b>About This Course .....</b>	<b>Preface-xxv</b>
Course Goals.....	Preface-xxv
Course Map.....	Preface-xxvi
Topics Not Covered.....	Preface-xxviii
How Prepared Are You?.....	Preface-xxix
Introductions .....	Preface-xxx
How to Use Course Materials .....	Preface-xxxii
Conventions .....	Preface-xxxii
Icons .....	Preface-xxxii
Typographical Conventions .....	Preface-xxxii
Additional Conventions.....	Preface-xxxiii
<b>Examining Object-Oriented Concepts and Terminology .....</b>	<b>1-1</b>
Objectives .....	1-1
Additional Resources .....	1-2
Examining Object Orientation.....	1-3
Software Complexity .....	1-4
Software Decomposition.....	1-6
Software Costs .....	1-7
Comparing the Procedural and OO Paradigms .....	1-8
Surveying the Fundamental OO Concepts .....	1-10
Objects.....	1-11
Classes .....	1-13
Abstraction.....	1-15
Encapsulation .....	1-17
Inheritance.....	1-19
Abstract Classes.....	1-21
Interfaces .....	1-23
Polymorphism .....	1-25
Cohesion.....	1-27
Coupling.....	1-29
Class Associations and Object Links.....	1-30
Delegation .....	1-32
Summary .....	1-35

<b>Introducing Modeling and the Software Development Process .....</b>	<b>2-1</b>
Objectives .....	2-1
Additional Resources .....	2-2
Exploring the OOSD Process.....	2-3
Describing Software Methodology.....	2-4
Listing the Workflows of the OOSD Process .....	2-6
Describing the Software Team Job Roles.....	2-7
Exploring the Requirements Gathering Workflow .....	2-9
Workflow Purpose and Job Roles.....	2-9
Workflow Activities and Artifacts.....	2-10
Exploring the Requirements Analysis Workflow .....	2-12
Workflow Purpose and Job Roles.....	2-12
Workflow Activities and Artifacts.....	2-13
Exploring the Architecture Workflow .....	2-15
Workflow Purpose and Job Roles.....	2-15
Workflow Activities and Artifacts.....	2-16
Exploring the Design Workflow .....	2-18
Workflow Purpose and Job Roles.....	2-18
Workflow Activities and Artifacts.....	2-19
Exploring the Implementation, Testing, and Deployment Workflows.....	2-21
Workflow Purpose and Job Roles.....	2-21
Workflow Activities and Artifacts.....	2-22
Examining the Benefits of Modeling Software .....	2-23
What Is a Model?.....	2-23
Why Model Software? .....	2-24
OOSD as Model Transformations .....	2-25
Defining the UML .....	2-26
Common UML Elements and Connectors .....	2-30
What UML Is and Is Not.....	2-36
Summary .....	2-39
<b>Creating Use Case Diagrams.....</b>	<b>3-1</b>
Objectives .....	3-1
Additional Resources .....	3-2
Process Map .....	3-3
Justifying the Need for a Use Case Diagram .....	3-4
Identifying the Elements of a Use Case Diagram.....	3-5
Actors .....	3-7
Use Cases.....	3-9
System Boundary .....	3-11
Use Case Associations .....	3-12
Creating the Initial Use Case Diagram .....	3-13
Identifying Additional Use Cases.....	3-15
Use Case Elaboration.....	3-16

Analyzing Inheritance Patterns .....	3-18
Analyzing Use Case Dependencies .....	3-20
Packaging the Use Case Views .....	3-24
Summary .....	3-26
<b>Creating Use Case Scenarios and Forms .....</b>	<b>4-1</b>
Objectives .....	4-1
Additional Resources .....	4-2
Process Map .....	4-3
Recording Use Case Scenarios .....	4-4
Selecting Use Case Scenarios.....	4-5
Writing a Use Case Scenario.....	4-6
Supplementary Specifications .....	4-10
Non-Functional Requirements (NFRs) .....	4-10
Glossary of Terms .....	4-11
Creating a Use Case Form .....	4-13
Description of a Use Case Form.....	4-13
Creating a Use Case Form .....	4-14
Summary .....	4-20
<b>Creating Activity Diagrams.....</b>	<b>5-1</b>
Objectives .....	5-1
Additional Resources .....	5-2
Process Map .....	5-3
Describing a Use Case With an Activity Diagram .....	5-4
Identifying the Elements of an Activity Diagram .....	5-5
Creating an Activity Diagram for a Use Case.....	5-13
Summary .....	5-15
<b>Determining the Key Abstractions.....</b>	<b>6-1</b>
Objectives .....	6-1
Additional Resources .....	6-2
Process Map .....	6-3
Introducing Key Abstractions .....	6-4
Identifying Candidate Key Abstractions .....	6-5
Identifying the Candidate Abstractions .....	6-6
Candidate Key Abstractions Form .....	6-8
Project Glossary .....	6-10
Discovering Key Abstractions Using CRC Analysis .....	6-11
Selecting a Key Abstraction Candidate .....	6-11
Identifying a Relevant Use Case .....	6-12
Determining Responsibilities and Collaborators .....	6-13
Documenting a Key Abstraction Using a CRC Card.....	6-15
Updating the Candidate Key Abstractions Form.....	6-16
Summary .....	6-19
<b>Constructing the Problem Domain Model.....</b>	<b>7-1</b>
Objectives .....	7-1

Additional Resources .....	7-2
Process Map .....	7-3
Introducing the Domain Model .....	7-4
Identifying the Elements of a Class Diagram .....	7-5
Class Nodes.....	7-6
Class Node Compartments.....	7-7
Associations .....	7-8
Creating a Domain Model .....	7-11
Step 1 – Draw the Class Nodes .....	7-12
Step 2 – Draw the Associations.....	7-13
Step 3 – Label the Association and Role Names.....	7-14
Step 4 – Label the Association Multiplicity .....	7-15
Step 5 – Draw the Navigation Arrows.....	7-16
Validating the Domain Model .....	7-16
Identifying the Elements of an Object Diagram .....	7-17
Object Nodes.....	7-18
Links.....	7-19
Validating the Domain Model Using Object Diagrams .....	7-20
Creating a Scenario Object Diagram .....	7-20
Comparing Object Diagrams to Validate the Domain Model.....	7-25
Summary .....	7-27

## **Transitioning from Analysis to Design**

<b>Using Interaction Diagrams .....</b>	<b>8-1</b>
Objectives .....	8-1
Additional Resources .....	8-2
Process Map .....	8-3
Introducing the Design Model.....	8-4
Interaction Diagrams .....	8-4
Comparing Analysis and Design.....	8-5
Robustness Analysis.....	8-6
Boundary Components .....	8-7
Service Components .....	8-8
Entity Components .....	8-9
Service and Entity Components.....	8-10
Boundary And Entity Components.....	8-10
Describing the Robustness Analysis Process .....	8-11
Identifying the Elements of a Communication Diagram .....	8-12
Creating a Communication Diagram.....	8-14
Step 1– Place the Actor in the Diagram .....	8-14
Step 2a – Identify Boundary Components .....	8-15
Step 2b – Identify Service Components .....	8-16
Step 2c – Identify Entity Components .....	8-17
Step 2d – Identify Additional Interactions .....	8-18
Communication Diagram Examples .....	8-19



Sequence Diagrams.....	8-21
Identifying the Elements of a Sequence Diagram .....	8-22
Fragments.....	8-24
Sequence Diagram Examples .....	8-25
Summary .....	8-28
<b>Modeling Object State Using State Machine Diagrams .....</b>	<b>9-1</b>
Objectives .....	9-1
Additional Resources .....	9-2
Process Map .....	9-3
Modeling Object State .....	9-4
Introducing Object State .....	9-4
Identifying the Elements of a State Machine Diagram.....	9-5
Creating a State Machine Diagram for a Complex Object .....	9-8
After Trigger Event.....	9-12
Self Transition.....	9-13
Junction.....	9-13
Junctions Example .....	9-14
Choice .....	9-15
Choice Example.....	9-16
Summary .....	9-17
<b>Applying Design Patterns to the Design Model.....</b>	<b>10-1</b>
Objectives .....	10-1
Additional Resources .....	10-2
Process Map .....	10-3
Explaining Software Patterns.....	10-4
Levels of Software Patterns .....	10-5
Design Principles .....	10-6
Describing the Composite Pattern.....	10-12
Composite Pattern: Problem .....	10-12
Composite Pattern: Solution.....	10-13
Composite Pattern: Consequences .....	10-14
Describing the Strategy Pattern .....	10-15
Strategy Pattern: Problem.....	10-15
Strategy Pattern: Solution .....	10-15
Strategy Pattern: Consequences.....	10-16
Describing the Observer Pattern.....	10-18
Observer Pattern: Problem .....	10-20
Observer Pattern: Solution.....	10-20
Observer Pattern: Consequences .....	10-21
Describing the Abstract Factory Pattern.....	10-22
Abstract Factory Pattern: Problem .....	10-23
Abstract Factory Pattern: Solution.....	10-23
Abstract Factory Pattern: Consequences .....	10-24
Programming a Complex Object .....	10-25

Problems With Coding a Complex Object .....	10-25
Describing the State Pattern .....	10-26
Summary .....	10-30
<b>Introducing Architectural Concepts and Diagrams.....</b>	<b>11-1</b>
Objectives .....	11-1
Additional Resources .....	11-2
Process Map .....	11-3
Justifying the Need for the Architect Role .....	11-4
Risks Associated With Large-Scale, Distributed Enterprise Systems .....	11-5
Quality of Service.....	11-7
Risk Evaluation and Control .....	11-8
The Role of the Architect.....	11-9
Distinguishing Between Architecture and Design.....	11-11
Architectural Principles .....	11-12
Architectural Patterns and Design Patterns.....	11-14
Tiers, Layers, and Systemic Qualities .....	11-16
Tiers.....	11-16
Layers.....	11-18
Systemic Qualities.....	11-19
Exploring the Architecture Workflow .....	11-20
Design Model.....	11-23
Architecture Template.....	11-24
Solution Model .....	11-25
Architectural Views .....	11-25
Describing Key Architecture Diagrams.....	11-26
Identifying the Elements of a Package Diagram .....	11-26
Identifying the Elements of a Component Diagram.....	11-29
Identifying the Elements of a Deployment Diagram.....	11-35
Selecting the Architecture Type.....	11-38
Standalone Applications .....	11-39
Client/Server (2-Tier) Applications .....	11-40
N-Tier Applications .....	11-41
Hotel Reservation System Architecture.....	11-44
Creating the Architecture Workflow Artifacts .....	11-45
Creating the Detailed Deployment Diagram .....	11-45
Creating the Architecture Template.....	11-47
Creating the Tiers and Layers Package Diagram .....	11-49
Summary .....	11-52
<b>Introducing the Architectural Tiers .....</b>	<b>12-1</b>
Objectives .....	12-1
Additional Resources .....	12-2
Process Map .....	12-3
Introducing the Client and Presentation Tiers .....	12-4
Boundary Interface Technologies .....	12-4

Overview of the GUI in the Client Tier of the Architecture Model.....	12-6
Overview of the Web UI in the Presentation Tier of the Architecture Model.....	12-15
Introducing the Business Tier .....	12-23
Exploring Distributed Object-Oriented Computing.....	12-23
Overview of the Detailed Deployment Diagram .....	12-30
Overview of the Tiers and Layers Package Diagram .....	12-31
Introducing the Resource and Integration Tiers .....	12-33
Exploring the Resource Tier .....	12-33
Exploring Integration Tier Technologies.....	12-43
Java™ Persistence API .....	12-50
Introducing the Solution Model.....	12-51
Overview a Solution Model for GUI Applications .....	12-52
Overview a Solution Model for WebUI Applications ....	12-55
Summary .....	12-58
<b>Refining the Class Design Model.....</b>	<b>13-1</b>
Objectives .....	13-1
Additional Resources .....	13-2
Process Map .....	13-3
Refining Attributes of the Domain Model.....	13-4
Refining the Attribute Metadata.....	13-4
Choosing an Appropriate Data Type.....	13-6
Creating Derived Attributes.....	13-7
Applying Encapsulation .....	13-7
Refining Class Relationships.....	13-8
Relationship Types .....	13-8
Navigation.....	13-12
Qualified Associations .....	13-13
Relationship Methods.....	13-14
Resolving Many-to-Many Relationships .....	13-15
Resolving Association Classes .....	13-17
Refining Methods.....	13-19
Annotating Method Behavior .....	13-21
Declaring Constructors .....	13-22
Reviewing the Coupling and Coherency of your Model .....	13-23
Reviewing Coupling.....	13-23
Reviewing Cohesion.....	13-24
Creating Components with Interfaces .....	13-25
Summary .....	13-27
<b>Overview of Software Development Processes .....</b>	<b>14-1</b>
Objectives .....	14-1
Additional Resources .....	14-2
Reviewing Software Methodology .....	14-3
Exploring Methodology Best Practices .....	14-4

Use-Case-Driven .....	14-4
Systemic-Quality-Driven .....	14-5
Architecture-Centric .....	14-5
Iterative and Incremental.....	14-6
Model-Based .....	14-8
Design Best Practices .....	14-10
Surveying Several Methodologies .....	14-11
Waterfall .....	14-11
Unified Software Development Process .....	14-13
Rational Unified Process .....	14-15
Scrum .....	14-15
eXtreme Programming.....	14-16
Choosing a Methodology.....	14-18
Choosing Waterfall .....	14-18
Choosing UP .....	14-19
Choosing RUP .....	14-19
Choosing Scrum .....	14-20
Choosing XP .....	14-21
Project Constraints and Risks.....	14-22
Project Constraints .....	14-22
Project Risks .....	14-23
Producing an Iteration Plan.....	14-29
Prioritizing Use Cases .....	14-29
Architectural Baseline .....	14-30
Timeboxing .....	14-31
80/20 Rule.....	14-31
Producing an Iteration Plan.....	14-32
Summary .....	14-35
<b>Overview of Frameworks .....</b>	<b>15-1</b>
Objectives .....	15-1
Additional Resources .....	15-2
Description of Frameworks .....	15-3
Using Existing Frameworks .....	15-5
Advantages and Disadvantages of Using Frameworks ....	15-5
Building Frameworks.....	15-7
Domain Neutral Frameworks .....	15-9
Advantages and Disadvantages of Building Frameworks.....	15-9
Summary .....	15-11
<b>Course Review .....</b>	<b>16-1</b>
Objectives .....	16-1
Overview .....	16-2
Reviewing Object Orientation.....	16-3
OO Concepts and Terminology: A Recap .....	16-5
Reviewing UML Diagrams.....	16-7

UML Diagrams: A Recap .....	16-9
Reviewing the Development Process.....	16-11
The Development Process: A Recap.....	16-13
The Analysis and Design Workflows: A Recap.....	16-14
Summary .....	16-18
<b>Drafting the Development Plan .....</b>	<b>A-1</b>
Objectives .....	A-1
Relevance.....	A-2
Additional Resources .....	A-3
Process Map .....	A-4
Describing How SunTone Architecture Methodology	
Relates to the Development Plan.....	A-5
Listing Needed Resources and Skills .....	A-7
Determining the Developers .....	A-7
Determining the Skill Set .....	A-9
Selecting Use Cases for Iterations.....	A-11
Criteria for Prioritizing Use Cases.....	A-11
Estimating Development Time for Use Cases .....	A-12
Determining the Duration of Each Iteration .....	A-15
Grouping Use Cases Into Iterations .....	A-16
Documenting the Development Plan.....	A-17
Summary .....	A-19
<b>Constructing the Software Solution .....</b>	<b>B-1</b>
Objectives .....	B-1
Process Map .....	B-2
Defining a Package Structure for the Solution.....	B-3
Using UML Packages .....	B-3
Applying Package Principles .....	B-3
Isolating Subsystems and Frameworks .....	B-10
Developing a Package Structure for the	
Solution Model .....	B-10
Mapping the Domain Model to Java Technology	
Class Code.....	B-13
Type Information .....	B-13
Attributes .....	B-14
Associations .....	B-16
Summary .....	B-24
<b>Testing the Software Solution.....</b>	<b>C-1</b>
Objectives .....	C-1
Process Map .....	C-2
Defining System Testing.....	C-3
Unit Tests.....	C-5
Integration Tests.....	C-6
Functional Tests.....	C-7

Developing a Functional Test.....	C-8
Identifying Functional Test Inputs.....	C-9
Identifying Functional Test Results.....	C-9
Identifying Functional Test Conditions.....	C-9
Creating Functional Test Variants.....	C-10
Documenting Test Cases.....	C-10
Summary .....	C-12
<b>Deploying the Software Solution.....</b>	<b>D-1</b>
Objectives .....	D-1
Process Map .....	D-2
Explaining System Deployment .....	D-3
Using the Deployment Diagram.....	D-4
Creating an Instance Deployment Diagram .....	D-5
Kiosk Application Instance Deployment Diagram.....	D-6
WebPresenceApp Instance Deployment Diagram.....	D-7
HotelApp Instance Deployment Diagram .....	D-8
Remote Application Server Instance Deployment Diagram.....	D-9
Summary .....	D-10
<b>Quick Reference for UML.....</b>	<b>E-1</b>
Additional Resources .....	E-2
UML Basics .....	E-3
General Elements .....	E-6
Packages .....	E-6
Stereotypes.....	E-8
Annotation .....	E-8
Constraints .....	E-9
Tagged Values .....	E-9
Use Case Diagrams .....	E-10
Class Diagrams.....	E-11
Class Nodes.....	E-11
Inheritance.....	E-14
Interface Implementation.....	E-15
Association, Roles, and Multiplicity .....	E-16
Aggregation and Composition .....	E-17
Association Classes.....	E-18
Other Association Elements .....	E-20
Object Diagrams.....	E-22
Communication Diagrams .....	E-24
Sequence Diagrams.....	E-26
State Machine Diagrams .....	E-28
Transitions.....	E-29
Activity Diagrams.....	E-30
Component Diagrams .....	E-35
Deployment Diagrams .....	E-37

---

<b>Additional Resources.....</b>	<b>F-1</b>
Additional Resources .....	F-1
<b>Glossary/Acronyms.....</b>	<b>Glossary-1</b>

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.



## List of Figures

---

Figure 1-1 Functional Decomposition .....	1-6
Figure 1-2 Object-Oriented Decomposition .....	1-6
Figure 1-3 An Example Object at Runtime .....	1-12
Figure 1-4 An Example Class .....	1-14
Figure 1-5 An Example of Good and Bad Abstractions .....	1-15
Figure 1-6 An Example of Encapsulation .....	1-17
Figure 1-7 An Example Inheritance Hierarchy .....	1-20
Figure 1-8 An Example Abstract Class .....	1-22
Figure 1-9 An Example Interface .....	1-24
Figure 1-10 An Example of Polymorphism .....	1-26
Figure 1-11 A Polymorphism Example Using Interfaces .....	1-26
Figure 1-12 An Example of Low and High Cohesion .....	1-28
Figure 1-13 Four Forms of Coupling .....	1-29
Figure 1-14 An Example of Object Associations .....	1-31
Figure 1-15 Issues with Inheritance on Non-Coherent Classes .....	1-33
Figure 1-16 Delegation to a more Cohesive Class .....	1-34
Figure 2-1 OOSD Hierarchy Pyramid .....	2-5
Figure 2-2 Software Development Job Roles .....	2-7
Figure 2-3 Activities and Artifacts of the Requirements Gathering Workflow .....	2-10
Figure 2-4 Activities and Artifacts of the Requirements Analysis Workflow .....	2-13
Figure 2-5 Activities and Artifacts of the Architecture Workflow .....	2-16
Figure 2-6 Activities and Artifacts of the Design Workflow .....	2-19
Figure 2-7 The Activities and Artifacts of the Implementation, Testing, and Deployment Workflows .....	2-22
Figure 2-8 Example Models From Traditional Architecture .....	2-23
Figure 2-9 Software Development as a Series of Model Transformations .....	2-25

Figure 2-10 Elements of UML Diagrams .....	2-27
Figure 2-11 Set of UML Diagrams .....	2-28
Figure 2-12 Views Created From UML Diagrams .....	2-30
Figure 2-13 Example of Using Packages .....	2-31
Figure 2-14 Example of Using a UML Note .....	2-32
Figure 2-15 Example of UML Dependencies and Stereotypes .....	2-33
Figure 3-1 Requirements Gathering Process Map .....	3-3
Figure 3-2 An Example Use Case Diagram .....	3-5
Figure 3-3 An Example Alternate Style Use Case Diagram .....	3-6
Figure 3-4 Actor Types .....	3-7
Figure 3-5 An Example Use Case .....	3-9
Figure 3-6 A Use Case Diagram With a System Boundary .....	3-11
Figure 3-7 A Use Case Diagram Without a System Boundary .....	3-11
Figure 3-8 An Example Use Case Association .....	3-12
Figure 3-9 Partial Initial Use Case Diagram for a Hotel .....	3-14
Figure 3-10 An Example High-Level Use Case .....	3-16
Figure 3-11 The High-Level Use Case Separated Into Individual Workflows .....	3-17
Figure 3-12 Example Actor Inheritance .....	3-18
Figure 3-13 Example Use Case Specialization .....	3-19
Figure 3-14 Example «include» Dependency .....	3-20
Figure 3-15 An «include» Dependency in the Hotel Reservation System .....	3-20
Figure 3-16 Example Where the Secondary Actor's Role is not Clear .....	3-21
Figure 3-17 Example Using «include» to Highlight the Secondary Actor's Participation .....	3-21
Figure 3-18 Example Extends Dependency .....	3-22
Figure 3-19 An «extend» Dependency in the Hotel Reservation System .....	3-22
Figure 3-20 A Combined Example From the Hotel Reservation System .....	3-23
Figure 3-21 Example Showing Packages of Use Case.....	3-24
Figure 3-22 Example of Marketing Use Cases and Actor .....	3-24
Figure 3-23 Example of Reservation Use Cases and Actors .....	3-25
Figure 4-1 Use Case Scenarios and Forms Process Map .....	4-3
Figure 5-1 Creating Activity Diagrams Process Map .....	5-3
Figure 5-2 Example of an Activity Diagram .....	5-5
Figure 5-3 An Example of Concurrent Activities .....	5-5
Figure 5-4 An Activity Diagram Represents Flow of Control .....	5-6
Figure 5-5 Branch and Merge Nodes .....	5-7

Figure 5-6 Iteration Loops in an Activity Diagram .....	5-7
Figure 5-7 Concurrent Flow in an Activity Diagram .....	5-8
Figure 5-8 Example of Object Flow between Activities .....	5-9
Figure 5-9 Example of Vertical Partitions in an Activity Diagram .....	5-10
Figure 5-10 Example of Send, Accept and Time Events. ....	5-11
Figure 5-11 Example of an Interruptible Activity Region .....	5-12
Figure 5-12 Activity Diagram showing a subset of the Create Reservation Use Case .....	5-13
Figure 5-13 Activity Diagram for the Create Reservation Use Case .....	5-14
Figure 6-1 Key Abstractions Process Map .....	6-3
Figure 6-2 A CRC Card Template .....	6-15
Figure 6-3 The CRC Card for the Reservation Key Abstraction .....	6-16
Figure 7-1 Domain Model Process Map .....	7-3
Figure 7-2 Elements of a UML Class Diagram .....	7-5
Figure 7-3 Different Ways to Represent a Class Node .....	7-6
Figure 7-4 An Example Class Node With Compartments .....	7-7
Figure 7-5 Example Association With Relationship and Role Labels .....	7-8
Figure 7-6 Example Association With Multiplicity Labels .....	7-8
Figure 7-7 Example Association With Navigation Arrows .....	7-10
Figure 7-8 Example Association Class .....	7-10
Figure 7-9 Step 1 – Draw the Class Nodes .....	7-12
Figure 7-10 Step 2 – Draw Associations Between Collaborating Classes .....	7-13
Figure 7-11 Step 3 – Record the Relationship and Role Names .....	7-14
Figure 7-12 Step 4 – Record the Association Multiplicity .....	7-15
Figure 7-13 Step 5 – Record the Association Navigation .....	7-16
Figure 7-14 Elements of a UML Object Diagram .....	7-17
Figure 7-15 Object Node Types .....	7-18
Figure 7-16 An Object Node With Attributes .....	7-18
Figure 7-17 An Object Diagram With Links .....	7-19
Figure 7-18 Step 1 – Create Reservation Scenario 1 .....	7-21
Figure 7-19 Step 2 – Create Reservation Scenario 1 .....	7-21
Figure 7-20 Step 3 – Create Reservation Scenario 1 .....	7-22
Figure 7-21 Step 4 – Create Reservation Scenario 1 .....	7-22
Figure 7-22 Step 5 – Create Reservation Scenario 1 .....	7-23
Figure 7-23 Create Reservation Scenario 2 .....	7-24
Figure 7-24 Revised Domain Model for the Hotel Reservation System .....	7-26
Figure 8-1 Interaction Diagrams Process Map .....	8-3
Figure 8-2 Introducing the Design Model .....	8-4

Figure 8-3 The Design Model Is Derived From the Requirements Model .....	8-6
Figure 8-4 An Example Boundary Component .....	8-7
Figure 8-5 An Example Service Component .....	8-8
Figure 8-6 An Example Entity Component .....	8-9
Figure 8-7 An example of coherent services .....	8-10
Figure 8-8 An example of a Boundary object accessing data from an Entity object .....	8-10
Figure 8-9 An Example of a Communication Diagram .....	8-12
Figure 8-10 A Variation Using Stereotype Icons .....	8-13
Figure 8-11 Step 1 – Place the Actor in the Communication Diagram .....	8-14
Figure 8-12 Step 2a – BookingAgent Interacts With the ReservationUI Component .....	8-15
Figure 8-13 Step 2b – The ReservationUI Component delegates some of its behavior to a business method in the ReservationService Component .....	8-16
Figure 8-14 Step 2c – The ReservationService Component creates the Reservation Entity Component .....	8-17
Figure 8-15 Additional Interactions between Services and Entity Components .....	8-18
Figure 8-16 Additional Interactions between Boundary and Entity Components .....	8-19
Figure 8-17 Example of a Primary (successful) Scenario Communication Diagram .....	8-20
Figure 8-18 An Example of a Secondary (unsuccessful) Scenario Communication Diagram .....	8-21
Figure 8-19 An Example Sequence Diagram .....	8-22
Figure 8-20 An Example Showing a Loop, an Alt, and a Reference Fragment .....	8-24
Figure 8-21 Example of a Primary (successful) and Secondary (unsuccessful) scenarios on the same diagram .....	8-25
Figure 8-22 Example of a Secondary Scenario Sequence Diagram .....	8-26
Figure 8-23 Example of a Sequence Diagram Fragment .....	8-27
Figure 9-1 Design Workflow Process Map .....	9-3
Figure 9-2 Example State Machine Diagram .....	9-5
Figure 9-3 HVAC State Machine Diagram With Transitions .....	9-6
Figure 9-4 Internal Structure of State Nodes .....	9-6
Figure 9-5 Complete HVAC State Machine Diagram .....	9-7
Figure 9-6 Step 1 – Start With the Initial and Final States .....	9-8
Figure 9-7 Step 2 – Determine Stable Object States .....	9-9
Figure 9-8 Step 3 – Specify the Partial Ordering of States .....	9-10
Figure 9-9 Step 4 – Specify the Transition Events and Actions .....	9-11
Figure 9-10 Step 5 – Specify the Actions Within a State .....	9-12

Figure 9-11 Stack Example Showing Self Transition .....	9-13
Figure 9-12 Order of Processing by Using Junction .....	9-14
Figure 9-13 Comparing Stacks With Junction and Without Junction .....	9-14
Figure 9-14 Order of Processing by Using Choice .....	9-15
Figure 9-15 Comparing Stacks With Choice and Without Choice .....	9-16
Figure 10-1 Design Workflow Process Map .....	10-3
Figure 10-2 GUI Example that Demonstrates Design Principles .....	10-6
Figure 10-3 Example of the Open Closed Principle .....	10-7
Figure 10-4 Using Inheritance to Provide Different GUI Layout Mechanisms .....	10-8
Figure 10-5 Variation in Multiple Features Leads to Combinatorial Explosion of Classes .....	10-9
Figure 10-6 Example of the Composite Reuse Principle .....	10-10
Figure 10-7 Dependency Inversion Principle Using an Abstract Class .....	10-11
Figure 10-8 Dependency Inversion Principle Using an Interface .....	10-11
Figure 10-9 An Example of the Composite Pattern in AWT .....	10-12
Figure 10-10 GoF Solution for the Composite Pattern .....	10-13
Figure 10-11 An Alternate Solution for the Composite Pattern .....	10-14
Figure 10-12 An Example of the Strategy Pattern in AWT .....	10-15
Figure 10-13 GoF Solution for the Strategy Pattern .....	10-16
Figure 10-14 An Example of the Observer Pattern in a GUI Application .....	10-18
Figure 10-15 Object Diagram of the Observer Example .....	10-19
Figure 10-16 Class Diagram of the Observer Example .....	10-19
Figure 10-17 GoF Solution for the Observer Pattern .....	10-20
Figure 10-18 An Example of the Abstract Factory Pattern in AWT .....	10-22
Figure 10-19 GoF Solution for the Abstract Factory Pattern .....	10-23
Figure 10-20 HVAC Class With Complex State Code .....	10-25
Figure 10-21 HVAC Class Using the State Pattern .....	10-26
Figure 10-22 HVAC Class Using the State Pattern .....	10-27
Figure 10-23 GoF Solution for the State Pattern .....	10-28
Figure 11-1 Architecture Process Map .....	11-3
Figure 11-2 Client-Server System .....	11-5
Figure 11-3 Highly Distributed Systems .....	11-6
Figure 11-4 Dependency Inversion Principle .....	11-13

Figure 11-5 SunTone Architecture Methodology	
3D Cube Diagram .....	11-16
Figure 11-6 Architecture Model in the OOSD Process .....	11-20
Figure 11-7 Example Design Components .....	11-23
Figure 11-8 Example Architecture Template .....	11-24
Figure 11-9 Example Solution Model .....	11-25
Figure 11-10 Elements of a UML Package Diagram .....	11-26
Figure 11-11 An Example Package Diagram .....	11-27
Figure 11-12 An Abstract Package Diagram .....	11-28
Figure 11-13 An Example Component Diagram .....	11-29
Figure 11-14 An Example of the UML 2	
Style Required Interface .....	11-29
Figure 11-15 An Example of a UML 2	
style Component Notation .....	11-30
Figure 11-16 Types of Software Components .....	11-32
Figure 11-17 Component Diagrams Can	
Show Software Dependencies .....	11-33
Figure 11-18 Component Diagram Showing	
a Web Service Dependency .....	11-33
Figure 11-19 Component Diagrams Can	
Represent Build Structures .....	11-34
Figure 11-20 An Example Deployment Diagram .....	11-35
Figure 11-21 An Example Instance Hardware Node .....	11-37
Figure 11-22 Generic Standalone Architecture Type .....	11-39
Figure 11-23 Generic Client/Server Architecture Type .....	11-40
Figure 11-24 Generic N-tier Architecture Type .....	11-41
Figure 11-25 Web-Centric N-Tier Architecture Type .....	11-42
Figure 11-26 Enterprise N-Tier Architecture Type .....	11-43
Figure 11-27 High-Level Deployment Diagram	
of the Hotel Reservation System .....	11-44
Figure 11-28 Example Detailed Deployment Diagram .....	11-46
Figure 11-29 Example Architecture Template .....	11-48
Figure 11-30 Tiers and Layers Diagram for	
the HotelApp .....	11-50
Figure 11-31 Tiers and Layers Diagram for	
the Hotel System's Web Presence .....	11-51
Figure 12-1 Architecture Tiers Process Map .....	12-3
Figure 12-2 Generic Application Components .....	12-6
Figure 12-3 Customer Management Screen .....	12-7
Figure 12-4 CustomerUI GUI Component Hierarchy .....	12-8
Figure 12-5 Java Technology Event Model .....	12-9
Figure 12-6 GUI Listeners as Controller Elements .....	12-10
Figure 12-7 MVC Pattern .....	12-11
Figure 12-8 MVC Component Types .....	12-12
Figure 12-9 Example Use of the MVC Pattern .....	12-13

Figure 12-10 A Partial Tiers and Layers Package Diagram for the HotelApp .....	12-14
Figure 12-11 Example Web Page Flow .....	12-16
Figure 12-12 Web UI Event Model .....	12-18
Figure 12-13 WebMVC Pattern .....	12-19
Figure 12-14 WebMVC Pattern Component Types .....	12-20
Figure 12-15 A Partial Tiers/Layers Package Diagram .....	12-22
Figure 12-16 Local Access to a Service Component .....	12-25
Figure 12-17 Applying the Dependency Inversion Principle .....	12-25
Figure 12-18 Class Diagram of the ResvService Interface and Implementation .....	12-26
Figure 12-19 An Abstract Version of Accessing a Remote Service .....	12-26
Figure 12-20 Accessing a Remote Service Without a Skeleton Component .....	12-27
Figure 12-21 RMI Uses Serialization to Pass Parameters .....	12-28
Figure 12-22 RMI Registry Stores Stubs for Remote Lookup .....	12-29
Figure 12-23 Example Detailed Deployment Diagram .....	12-30
Figure 12-24 Tiers and Layers Diagram for the HotelApp .....	12-31
Figure 12-25 Tiers and Layers Diagram for the WebPresenceApp .....	12-32
Figure 12-26 A Simplified Domain Model of the Hotel Reservation System .....	12-36
Figure 12-27 Step 1 – Creating Entity Tables .....	12-37
Figure 12-28 Step 2 – Specifying Primary Keys .....	12-38
Figure 12-29 Step 3 – Creating One-to-Many Relationships .....	12-39
Figure 12-30 Step 3 – Creating a Many-to-Many Resolution Table .....	12-40
Figure 12-31 Example Detailed Deployment Diagram .....	12-41
Figure 12-32 Partial HRS Tiers and Layers Package Diagram .....	12-42
Figure 12-33 DAO Pattern Example 1 .....	12-45
Figure 12-34 DAO Pattern Example 2 .....	12-46
Figure 12-35 The DAO Pattern in a Detailed Deployment Diagram .....	12-47
Figure 12-36 Example Detailed Deployment Diagram .....	12-48
Figure 12-37 Complete HRS Tiers/Layers Package Diagram .....	12-49
Figure 12-38 Introducing the Solution Model .....	12-51
Figure 12-39 A Complete Analysis Model for the Create a Reservation Use Case .....	12-52
Figure 12-40 A Complete Solution Model for the Create a Reservation Use Case .....	12-54

Figure 12-41 A Complete Analysis Model for the Create a Reservation Online Use Case .....	12-56
Figure 12-42 A Complete Solution Model for the Create a Reservation Online Use Case .....	12-57
Figure 13-1 Class Design Workflow Process Map .....	13-3
Figure 13-2 Example Refinement of Attributes .....	13-5
Figure 13-3 Example Derived Attribute .....	13-7
Figure 13-4 Example Use of Encapsulation .....	13-7
Figure 13-5 An Association Example .....	13-9
Figure 13-6 An Aggregation Example .....	13-10
Figure 13-7 A Composition Example .....	13-11
Figure 13-8 The Three Forms of Navigation Indicators .....	13-12
Figure 13-9 An Example Navigation Refinement .....	13-13
Figure 13-10 An Example Qualified Association .....	13-13
Figure 13-11 Association Methods for a One-to-One Relationship .....	13-14
Figure 13-12 Association Methods for a One-to-Many Relationship .....	13-15
Figure 13-13 Drop the Many-to-Many Relationship .....	13-16
Figure 13-14 Introduce an Intermediate Relationship .....	13-16
Figure 13-15 Resolving Association Classes: Case One .....	13-17
Figure 13-16 Resolving Association Classes: Case Two .....	13-18
Figure 13-17 Example Refinement of Methods .....	13-20
Figure 13-18 Example of UML Method Annotation .....	13-21
Figure 13-19 Example Constructors .....	13-22
Figure 13-20 Example Comparing High and Low Coupling .....	13-23
Figure 13-21 Example Comparing High and Low Cohesion .....	13-24
Figure 13-22 Example of Components with Required and Provided Interfaces .....	13-25
Figure 13-23 Example of Components with Required and Provided Interfaces and Dependency Arrows .....	13-26
Figure 14-1 Software Development Methodology .....	14-3
Figure 14-2 Waterfall Methodology .....	14-12
Figure 14-3 UP Methodology .....	14-14
Figure 14-4 Rational's Tool Set for the RUP Methodology .....	14-15
Figure 14-5 Scrum Development Framework .....	14-16
Figure 14-6 XP Methodology .....	14-17
Figure 14-7 Sample Library Use Case Diagram .....	14-32
Figure 15-1 Example showing framework of abstract classes .....	15-8
Figure 15-2 Example showing a framework of interfaces and abstract classes .....	15-8
Figure 16-1 UML Diagrams .....	16-10



Figure 16-2 Example of a High-Level View of the Analysis, Design, and Architecture Workflows .....	16-14
Figure 16-3 Example of Activities in the Analysis Workflow .....	16-15
Figure 16-4 Example of the Identify Key Abstractions Activity .....	16-16
Figure 16-5 Example of Activities in the Design Workflow .....	16-17
Figure A-1 Implementation Workflow Process Map.....	A-4
Figure A-2 Iterations for the Hotel Reservation System .....	A-16
Figure A-3 Two Variations on the UML Package Icon.....	A-17
Figure A-4 Package Diagram of the Elaboration Phase for the Hotel Reservation System.....	A-18
Figure B-1 Implementation Workflow Process Map .....	B-2
Figure B-2 Example Code for a Java Technology Package .....	B-3
Figure B-3 An Example Package Dependency .....	B-4
Figure B-4 An Example Violation of the ADP .....	B-6
Figure B-5 An Example Refactoring to Achieve ACP .....	B-7
Figure B-6 An Example of the SDP for the Hotel Reservation System .....	B-8
Figure B-7 An Example of the SAP for the Hotel Reservation System .....	B-9
Figure B-8 HRS Package Dependencies Grouped by Tiers .....	B-11
Figure B-9 HRS Package Dependencies Between the Application and Virtual Platform Layers .....	B-12
Figure B-10 Java Technology Code for Three Types of Classes .....	B-13
Figure B-11 Java Technology Code for Class Inheritance .....	B-14
Figure B-12 Java Technology Code for Interface Implementation.....	B-14
Figure B-13 Java Technology Code for Attribute Declarations .....	B-15
Figure B-14 Java Technology Code for Class Scoped Attributes .....	B-16
Figure B-15 Java Technology Code for an Association .....	B-16
Figure B-16 Java Technology Code for a Unidirectional Association .....	B-17
Figure B-17 Java Technology Code for a Bidirectional Association .....	B-17
Figure B-18 Java Technology Code for Association Methods .....	B-18
Figure B-19 Java Technology Code for a One-to-Many Association .....	B-19
Figure B-20 Java Technology Code for an Indexed Qualified Association.....	B-20

Figure B-21 Java Technology Code for a Symbolic Qualified Association .....	B-21
Figure B-22 Java Technology Code for an Aggregation.....	B-22
Figure B-23 Java Technology Code for a Composition .....	B-23
Figure C-1 Testing Workflow Process Map .....	C-2
Figure D-1 Deployment Workflow Process Map .....	D-2
Figure D-2 Hotel Reservation Descriptive Deployment Diagram .....	D-5
Figure D-3 Instance Deployment Diagram for the KioskApp .....	D-6
Figure D-4 The Instance Deployment Diagram for the WebPresenceApp .....	D-7
Figure D-5 The Instance Deployment Diagram for the HotelApp .....	D-8
Figure D-6 Instance Deployment Diagram for the Business Tier Host .....	D-9
Figure E-1 Example Package .....	E-6
Figure E-2 Example Stereotype .....	E-8
Figure E-3 Example Annotation .....	E-8
Figure E-4 Example Constraints .....	E-9
Figure E-5 Example Tagged Values .....	E-9
Figure E-6 An Example Use Case Diagram .....	E-10
Figure E-7 Several Class Nodes .....	E-11
Figure E-8 Elements of a Class Node .....	E-12
Figure E-9 An Example Class Node With Static Elements .....	E-13
Figure E-10 Class Inheritance Relationship.....	E-14
Figure E-11 An Example of a Class Implementing an Interface.....	E-15
Figure E-12 Class Associations and Roles .....	E-16
Figure E-13 Example Aggregation and Composition.....	E-18
Figure E-14 A Simple Association Class.....	E-18
Figure E-15 A More Complex Association Class.....	E-19
Figure E-16 Other Associations Elements .....	E-20
Figure E-17 An Example Object Diagram .....	E-22
Figure E-18 An Example Object Diagram .....	E-23
Figure E-19 An Example User-driven Communication Diagram .....	E-24
Figure E-20 Another Communication Diagram .....	E-25
Figure E-21 An Example User-driven Sequence Diagram.....	E-26
Figure E-22 Another Sequence Diagram .....	E-27
Figure E-23 An Example State Machine Diagram.....	E-28
Figure E-24 An Example State Transition .....	E-29
Figure E-25 Activities and Transitions.....	E-30
Figure E-26 Branching and Looping .....	E-31
Figure E-27 An Example Activity Diagram .....	E-32

---

Figure E-28 An Example Activity Diagram Showing UML2 Object Flow Pins .....	E-34
Figure E-29 Example Component Nodes .....	E-35
Figure E-30 An Example Component Diagram.....	E-35
Figure E-31 A Component Diagram With Interfaces .....	E-36
Figure E-32 An Example Deployment Diagram.....	E-37

Adolfo De+la+Rosa (adolfoelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# About This Course

---

## Course Goals

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

- Apply object-oriented (OO) technologies to meet your software requirements
- Create proportionate and appropriate Unified Modeling Language (UML) models or text models at each stage in the software development process
- Analyze system requirements using use cases to determine the analysis (business domain) model
- Create analysis models that capture the business requirements of the system
- Explain how to fit the design components into the chosen architecture
- Create design (solution) models that support requirements of the system
- Apply the patterns and principles used in analysis and design
- Describe common Object-Oriented Software Development (OOSD) processes

# Course Map

The following course map enables you to see what you have accomplished and where you are going in reference to the course goals.

## Introduction to Object Orientation, UML and the Software Development Process

Examining  
Object-Oriented  
Concepts and Terminology

Introducing Modeling  
and the Software  
Development Process

## Object-Oriented Analysis

Creating Use Case  
Diagrams

Creating Use Case  
Scenarios and Forms

Creating  
Activity Diagrams

Determining the  
Key Abstractions

Constructing the  
Problem Domain Model

## Object-Oriented Design and Architecture

Transitioning from  
Analysis to Design Using  
Interaction Diagrams

Modeling Object State  
Using State Machine  
Diagrams

Applying  
Design Patterns  
to the Design Model

Introducing Architectural  
Concepts and Diagrams

Introducing the  
Architectural Tiers

Refining the Class  
Design Model

## Object-Oriented Development Process and Frameworks

Overview of Software  
Development  
Processes

Overview of  
Frameworks

**Course Review**

Course Review

**Construct, Test, and Deploy the System Solution\***

Drafting the  
Development Plan

Constructing the  
Software Solution

Testing the  
Software Solution

Deploying the  
Software Solution

\*These modules are appendices.

## Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Services:

- Fundamental Java technology – Covered in SL-275-SE6: *Java™ Programming Language*
- Enterprise edition Java technology – Covered in FJ-310-EE5: *Developing Applications for the Java™ EE Platform*

Refer to the Sun Services catalog for specific information and registration.



## How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Do you have a general understanding of a programming language?
- Do you have an understanding of the fundamentals of the software system development process?

# Introductions

Now that you have been introduced to the course, introduce yourself to the other students and the instructor, addressing the following items:

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to requirements gathering and analysis
- Experience related to software architecture and design
- Experience related to using a software development process
- Experience related to modeling notations, such as Object Modeling Technique (OMT) or Unified Modeling Language (UML)
- Reasons for enrolling in this course
- Expectations for this course

# How to Use Course Materials

To enable you to succeed in this course, these course materials contain a learning module that is composed of the following components:

- Goals – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- Objectives – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- Lecture – The instructor presents information specific to the objective of the module. This information helps you learn the knowledge and skills necessary to succeed with the activities.
- Activities – The activities take on various forms, such as an exercise, self-check, discussion, and demonstration. Activities help you facilitate the mastery of an objective. The majority of the activities are designed to be performed in small groups.
- Visual aids – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

# Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

## Icons



**Additional resources** – Indicates other references that provide additional information on the topics described in the module.



**Discussion** – Indicates a small-group or class discussion on the current topic is recommended at this time.



**Note** – Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

## Typographical Conventions

Courier is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

```
Use ls -al to list all files.
system% You have mail.
```

Courier is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

```
The getServletInfo method is used to get author information.
The java.awt.Dialog class contains Dialog constructor.
```

**Courier bold** is used for characters and numbers that you type; for example:

To list the files in this directory, type:

```
# ls
```

**Courier bold** is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

Notice the `javax.servlet` interface is imported to allow access to its life cycle methods (Line 2).

*Courier italics* is used for variables and command-line placeholders that are replaced with a real name or value; for example:

To delete a file, use the `rm filename` command.

***Courier italic bold*** is used to represent variables whose values are to be entered by the student as part of an activity; for example:

Type `chmod a+rw filename` to grant read, write, and execute rights for *filename* to world, group, and users.

*Palatino italics* is used for book titles, new words or terms, or words that you want to emphasize; for example:

Read Chapter 6 in the *User's Guide*.  
These are called *class* options.

## Additional Conventions

Java™ programming language examples use the following additional conventions:

- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:  
“The `doIt` method...” refers to any method called `doIt`.  
“The `doIt()` method...” refers to a method called `doIt` that takes no arguments.
- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.

- If a command used in the Solaris™ Operating Environment is different from a command used in the Microsoft Windows platform, both commands are shown; for example:

If working in the Solaris Operating Environment

```
> cd SERVER_ROOT/bin
```

If working in Microsoft Windows

```
C:> cd SERVER_ROOT\bin
```

# Examining Object-Oriented Concepts and Terminology

---

## Objectives

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

- Describe the important object-oriented (OO) concepts
- Describe the fundamental OO terminology

## Additional Resources



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

- Rumbaugh, James, Jacobson Ivor, Booch Grady. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004.
- Larman, Craig. *Applying UML and Patterns (3rd ed)*. Prentice Hall, 2005.
- Fowler, Martin. *UML Distilled (2nd ed)*. Addison-Wesley, 2000.
- Booch, Grady. *Object-Oriented Analysis and Design with Applications (2nd ed)*. The Benjamin/Cummins Publishing Company, Inc., Redwood City, 1994.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, 2000.
- Meyer, Bertrand. *Object-Oriented Software Construction (2nd ed)*. Prentice Hall, Upper Saddle River, 1997.
- Pressman, Roger. *Software Engineering A Practitioner's Approach, Fifth edition*. McGraw-Hill, 2001.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.



# Examining Object Orientation

OO concepts affect the whole development process starting with gathering requirements, domain modeling, design, and implementation. The following are considerations:

- Humans think in terms of nouns (objects) and verbs (behaviors of objects).

Human languages and thought patterns are formed around objects in the world, such as people, places, and things, and around the actions that manipulate the world of objects. OO principles put our focus on objects and what actions these objects perform (their so called responsibilities). Objects can also collaborate with other objects to perform an action.

- With OOSD, both problem and solution domains are modeled using OO concepts.

In OOSD, the development team is asked to create a software system that supports a *business process*. These processes are modeled with objects, responsibilities, and collaborations, because these notions fit our mental model of the *business process*.

---

**Note** – By business process, we mean a process that can have a wider scope than the software system.

---



- The *Unified Modeling Language* (UML) is a de facto standard for modeling OO software.

Although not essential, it is useful to be able to visualize OO concepts in terms of views and models. As a consequence, a de facto graphical notation called the UML has evolved. The UML was designed to model many OO aspects, including *objects*, *responsibilities*, and *collaborations*. Therefore, the UML is a good tool for visualizing our mental models.

- OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.

Various OO languages provide language syntax (such as class definitions) and semantic structures (such as runtime objects) based on OO principles. The UML has mappings for many mainstream OO languages, such as the Java programming language and C++. Therefore, the UML is a good bridge to implementation language

and the implementation language enables you to implement the system in a manner closer to the mental models of the problem and solution space.

“Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the object types than on the actions.” (Meyer page vi)

OO concepts affect the following issues:

- Software complexity
- Software decomposition
- Software costs

## Software Complexity

Complex systems have the following characteristics:

- They have a *hierarchical structure*.  
In procedural programming, a complex system is decomposed into a hierarchy of subroutines. This structure tends to be brittle.  
In OO programming, a complex system is decomposed into a hierarchy of collaborating objects. This structure tends to be more flexible and extensible because you are recomposing different collaborations between existing objects to solve new problems.
- The choice of *which components are primitive* in the system is arbitrary.  
The choice of what is primitive in a system depends on the perspective of the observer. This principle provides the designer with a handle on system complexity by shielding the complex details of one layer behind an abstraction (a method or object) on a higher level.  
This principle also affects modeling. You do not have to show all of the system complexity in one view; rather, you can create different views at different levels of abstraction.

- A system can be split by intra- and inter-component relationships. This *separation of concerns* enables you to study each part in relative isolation.

Separation of concerns is an important principle in designing and building complex systems. Using this principle, an object should focus its activity on one or a few simple concerns. The client of that object does not need to know how the object does its work.

For example, a UI component deals with a user's actions, such as processing a click to a button or verifying that a data field has a legitimate value. A business logic component does not need to know *how* the data is entered when it processes some operation.

- Complex systems are usually composed of only a *few types of components in various combinations*.

Like organic systems, software systems tend to have few fundamental types of components that are used to build the system. In animal biology, these structures include skin, bones, muscle, neurons, and so on. In computer systems, these structures include user interface, control (workflow), and entity components.

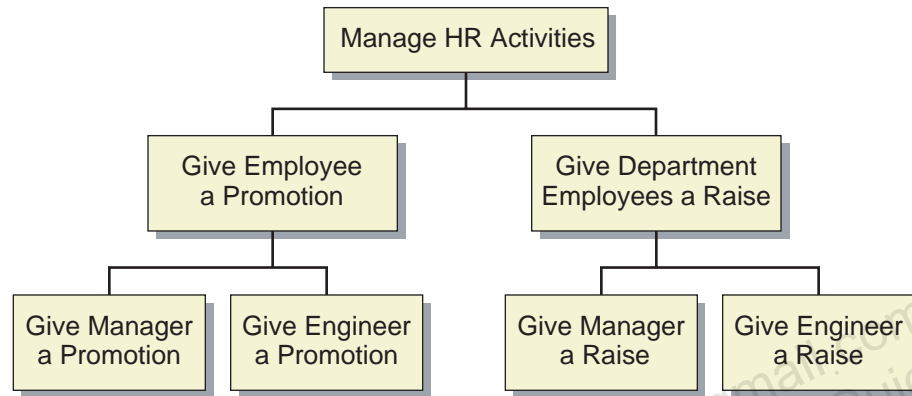
- A successful, complex system invariably *evolves from a simple working system*.

Successful complex systems tend to be grown rather than created. A complex system usually starts small with a few domain objects and adds complexity by extending these domain objects and rearranging them in new ways.

For example, the web application for Amazon.com started by selling books, and then started selling CDs and videos, and providing reader reviews, promotions, and so on.

## Software Decomposition

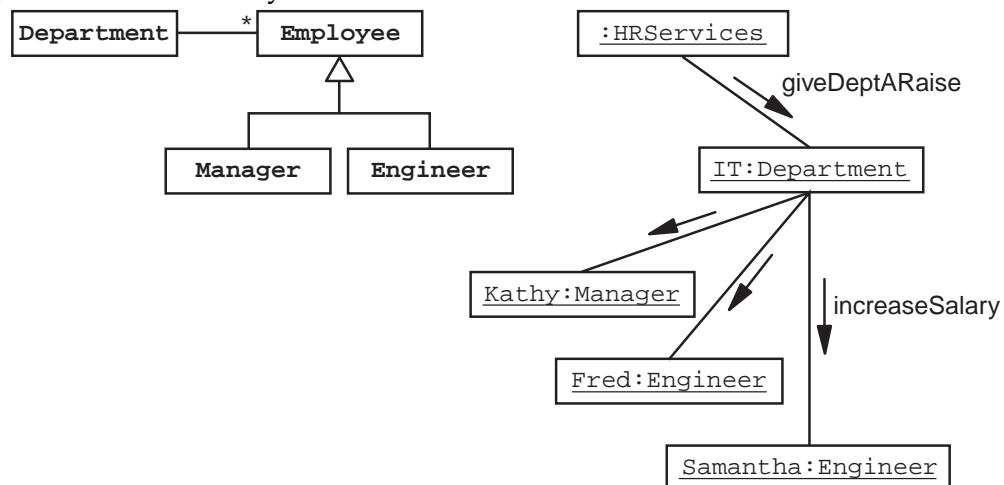
In procedural programming, tasks are decomposed into a hierarchy of procedures. Figure 1-1 shows an example hierarchy of procedures for managing human resources (HR) tasks, such as giving a promotion or a raise.



**Figure 1-1** Functional Decomposition

This decomposition seems natural until you try to extend the program. For example, how would you support a new type of employee, such as a lab technician? If this type of employee requires unique actions for “give a raise” and “give a promotion,” then the procedural hierarchy must be modified in several places. This is the definition of brittle.

Alternatively, using OO principles, a system is composed of a hierarchy of collaborating objects. Figure 1-2 shows an example object hierarchy. On the right, the department object delegates to the employee objects to perform a “give a raise” action. Each employee object has a type defined by the class hierarchy on the left.



**Figure 1-2** Object-Oriented Decomposition

To add a new type of employee, the system only needs to be extended by adding a new subclass of `Employee` with the code necessary to “give a raise.” Adding this class does not require any changes to the `Department` class which executes the `increaseSalary` method on any type of employee. This makes OO designs more flexible and resilient to change.

## Software Costs

OO principles can reduce development costs in the following ways:

- OO principles provide a natural technique for modeling business entities and processes from the early stages of a project.

OO development is an improvement over procedural development because you can model software that maps closely to your mental models. This relationship between mental models and software models increases productivity.

- OO-modeled business entities and processes are easier to implement in an OO language.

Building software in an OO language also increases productivity by enabling the programmers to write code that maps closely to the design models.

OO principles can reduce maintenance costs in the following ways:

- Changeability, flexibility, and adaptability of software is important to keep software running for a long time.

Meyer (page 17) states that “it is widely estimated that 70 percent of the cost of software is devoted to maintenance,” and that the bulk of the maintenance effort is in changes to user requirements and changes in data formats.

OO designs tend to be more flexible and adaptable than procedural designs; thus reducing maintenance costs.

- OO-modeled business entities and processes can be adapted to new functional requirements.

OO designs are easier to change in response to new business requirements. OO designs focus on identifying stable business objects that can be made to interact in new ways.

## Comparing the Procedural and OO Paradigms

Many programming paradigms have risen over the past half-century. These include: assembler (machine language), procedural, OO, functional, and rule-based to name only a few. Procedural and OO are the leading paradigms in the software industry.

This course focuses on the OO paradigm because this paradigm (which subsumes the procedural paradigm) has significant benefits over a strict procedural paradigm. These paradigms are compared in Table 1-1.

**Table 1-1** Comparing the Procedural and OO Paradigms

	Procedural Paradigm	OO Paradigm
Organizational structure	<ul style="list-style-type: none"> <li>Focuses on hierarchy of procedures and subprocedures</li> <li>Data is separate from procedures</li> </ul>	<ul style="list-style-type: none"> <li>Network of collaborating objects</li> <li>Methods (processes) are often bound together with the state (data) of the object</li> </ul>
Protection against inappropriate modification or access	Data is difficult to protect against inappropriate modifications or access when it is passed to or referenced by many different procedures.	The state (data) as well as methods of objects can be protected against inappropriate modifications or access by using encapsulation. Therefore, an object can protect its own state.
Ability to modify software	Can be expensive and difficult to make software that is easy to change, resulting in many “Brittle” systems with software that is difficult to change	Robust software that is easy to change, if written using good OO principles and patterns
Reuse	Reuse of methods is often achieved by copy-and-paste or 1001 parameters.	Reuse of code by using generic components (one or more objects) with well-defined interfaces. This is achieved by extension of classes (or interfaces) or by composition of objects.

**Table 1-1** Comparing the Procedural and OO Paradigms

	<b>Procedural Paradigm</b>	<b>OO Paradigm</b>
Configuration of special cases	Often requires if or switch statements. Modification is risky because it often requires altering existing code. So, modifications must be done with extreme care apart from requiring extensive regression testing. These factors make even minor changes costly to implement.	Polymorphic behavior can facilitate the possibility of modifications being primarily additive, subtractive, or substitution of whole components (object or group of objects); thereby, reducing the associated risks and costs.

## Surveying the Fundamental OO Concepts

In this section, you will survey the following fundamental OO concepts:

- *Objects*
- *Classes*
- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Abstract classes*
- *Interfaces*
- *Polymorphism*
- *Cohesion*
- *Coupling*
- *Class associations and object links*
- *Delegation*

There are many other concepts and terminologies that will be covered throughout the course, but these concepts and terminologies are essential for you to understand before this course describes the OOSD disciplines.



## Objects

object = state + behavior

“An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class.”  
(Booch Object Solutions page 305)

Objects are runtime features of an OO system. Objects:

- Have identity

Every object at runtime has an identity that is unique and is independent of its attribute values. The internal representation of object identity is usually hidden by the OO language. However, in C++ an object's identity is its pointer (address in physical memory). In the Java programming language an object's identity is hidden from the programmer; however, program variables refer to objects. Therefore, object references can be passed between objects and methods.

- Are an instance of only one class

A class defines a type of object. At runtime, an object is defined by one class. However, that class might be an extension of other classes as a result of using inheritance (which is covered later in this section).

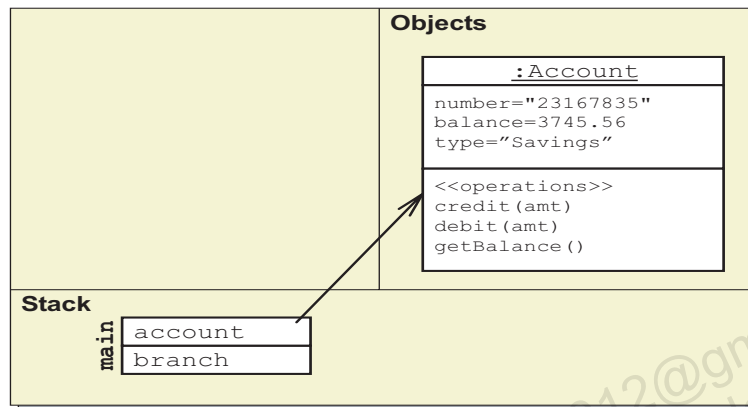
- Have attribute values that are unique to that object

Every object of the same class has the same set of attributes, *but* the values of these attributes are unique for every object. This means that you can change an attribute of one Account object without affecting any other Account object in memory.

- Have methods that are common to the class

The operations (or actions) of an object are defined by the set of methods that are implemented in the object's class or in an ancestor class (inheritance is covered later in this section).

Figure 1-3 illustrates an example Account object. The outer boxes represent aspects of a program's runtime environment. The Stack box (lower rectangle) represents the current execution frame (a stack frame) with program variables for that frame. In this example, the account variable holds a reference to the Account object. The Objects box (upper right) represents the space in runtime memory that holds objects; this is usually called the "heap."



**Figure 1-3** An Example Object at Runtime

## Classes

A class is a blueprint or prototype from which objects are created.  
(The Java™ Tutorials)

In OO programming, a class is usually a language construct that forms the common structure and behavior from which objects are instantiated.

Classes provide the following features:

- The metadata for attributes

The class does not hold data; the object holds the data. However, the class maintains the metadata of each attribute. This metadata includes the data type and the initial value (if specified).




---

**Note** – Most OO languages also support attributes that are scoped on the class itself. Class-scoped attributes are shared among all instances of that class.

---

- The signature for methods

The operations or actions of an object are usually called methods in OO terminology. Methods have two elements: signature and implementation. The signature includes the name of the method, the list of parameters (parameter name and type), and the return type.

- The implementation of the methods (usually)

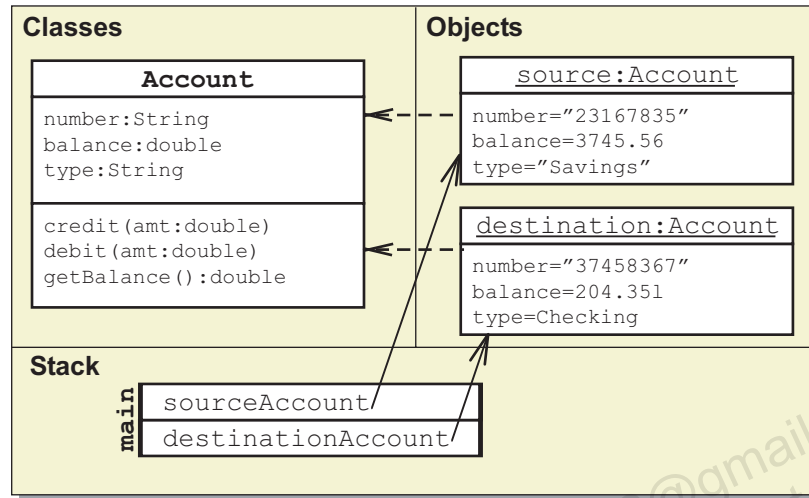
The implementation of a method is the set of programming statements that specify how the operation is to be performed.

In some OO languages you can declare a method signature and not provide an implementation; this is called an *abstract method*. If a class has one or more abstract methods, then it is called an *abstract class*. Abstract classes are used in design to force a subclass to implement the abstract methods. Both the C++ and Java programming languages support abstract methods.

- The constructors to initialize attributes at creation time

A constructor is a set of instructions that initializes an instance. Both the C++ and Java programming languages support constructors. Figure 1-4 illustrates an example Account class. Notice that the

operations have now been placed with the class rather than with the object. Classes hold methods and objects hold data. Objects execute methods by looking up the method implementation in the class.



**Figure 1-4** An Example Class

**Note** – We have kept the data types of the attributes simple for now.



**Discussion** – Discuss how you might do a transfer of funds from sourceAccount to destinationAccount

# Abstraction

Abstraction is “something that summarizes or concentrates the essentials of a larger thing” (Webster New Collegiate Dictionary)

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

Abstraction is a fundamental OO concept because finding the right abstractions enables you to design just those features defined by the system requirements and to ignore all other details that are irrelevant to the problem being solved.

Figure 1-5 illustrates two example Engineer classes. The class on the left has only the set of attributes and operations needed to satisfy the system requirements. The class on the right has a few attributes and objects that exist for most engineers (such as fingers and toes), but are irrelevant to the system requirements.

Engineer	Engineer
fname:String lname:String salary:Money	fname:String lname:String salary:Money fingers:int toes:int hairColor:String politicalParty:String
increaseSalary(amt) designSoftware() implementCode()	increaseSalary(amt) designSoftware() implementCode() eatBreakfast() brushHair() vote()

**Figure 1-5** An Example of Good and Bad Abstractions

Another aspect of abstraction in software is the idea that you can hide implementation details behind a public interface (one or more methods). This is often called *information hiding* and is discussed in “Encapsulation” on page 1-17.

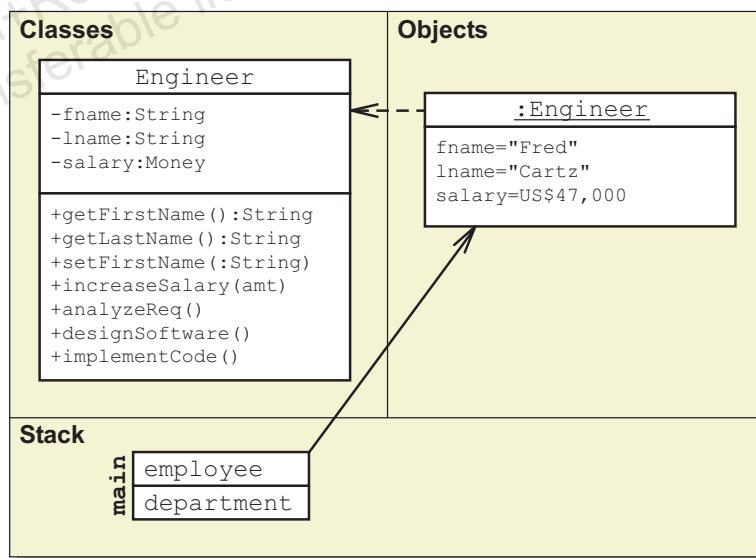
## Encapsulation

Encapsulation means “to enclose in or as if in a capsule.” (Webster New Collegiate Dictionary)

Encapsulation is an essential property of an object. An object, as a programming construct, is a capsule that holds the object’s internal state within its boundary. Most OO languages, such as the C++ and Java programming languages, have encapsulation built into the language itself.

In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “hide implementation details behind a set of non-private methods.” It is this aspect of encapsulation that most programmers think of, because in order to create proper encapsulation you must do two things: Make all attributes private and provide accessor and mutator methods to provide an abstract interface to the data held within the object capsule.

Figure 1-6 illustrates an example of encapsulation using the Engineer class. Notice that the attributes have been declared private (indicated by the minus sign before the attribute name) and accessor methods (such as `getFirstName()`) and mutator methods (such as `setFirstName()`) have been declared.



✗ <code>name = employee.fname;</code>	✓ <code>name = employee.getFirstName();</code>
✗ <code>employee.fname = "Samantha";</code>	✓ <code>employee.setFirstName("Samantha");</code>

**Figure 1-6** An Example of Encapsulation

Some programmers might argue that creating these methods is a waste of time. They might say: “Why do I need a method that just returns the value of an attribute?” or “Why do I need a method to set an attribute?” The following reasons might help relieve these concerns:

- Limiting the scope of bugs  
Strong encapsulation reduces the number of bugs in which one class incorrectly uses another class. For example, suppose that you are using a database to store employee records. Further, suppose that the first name field in the database has been defined as a `VARCHAR(10)`. This constraint means that the program must not try to store a string of length greater than 10 characters. If you did not use encapsulation, then a client of the `Engineer` class could set the `fname` attribute to a string of any length. With proper encapsulation, the `setFirstName` method could perform the length verification before setting the value. If the length check fails, then the mutator method could throw an exception.
- Abstraction  
To use an encapsulated class, you only need to know the purpose and signature of the public methods of the class. This tactic enables the programmer of the class to change the implementation inside the class without changing the public interface. The client of the class would never know that anything had changed.



## Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.” (Meyer page 1197)

Features of inheritance:

- Attributes and methods from the superclass are included in the subclass.

Because managers and engineers share some common attributes (name and salary), a superclass `Employee` is defined to hold these common attributes and operations. The `Engineer` and `Manager` subclasses inherit all of the features of the superclass plus these classes can extend the superclass with new features specific to that type of employee.

- Subclass methods can override superclass methods.

Notice that the `increaseSalary` method is declared by the `Employee` class. This declaration means that the subclasses will inherit this default behavior. However, the `Manager` class also includes the `increaseSalary` method. This declaration means that an object of type `Manager` will use its own implementation to “get a raise.”

- The following conditions must be true for the inheritance relationship to be plausible:

- A subclass object *is a* (is a kind of) the superclass object.

Using the *is a* or *is a kind of* test helps you avoid using the inheritance relationship when it is either inappropriate now or in the future (if changes are made). For instance, *is a Product* an `Employee` fails the test, whereas *is a Engineer* an `Employee` passes the test. However, even if a relationship passes the test, it does not necessarily mean that using the inheritance relationship is the best approach. We will discuss these issues later in this module.

- Inheritance should conform to Liskov’s Substitution Principle (LSP)

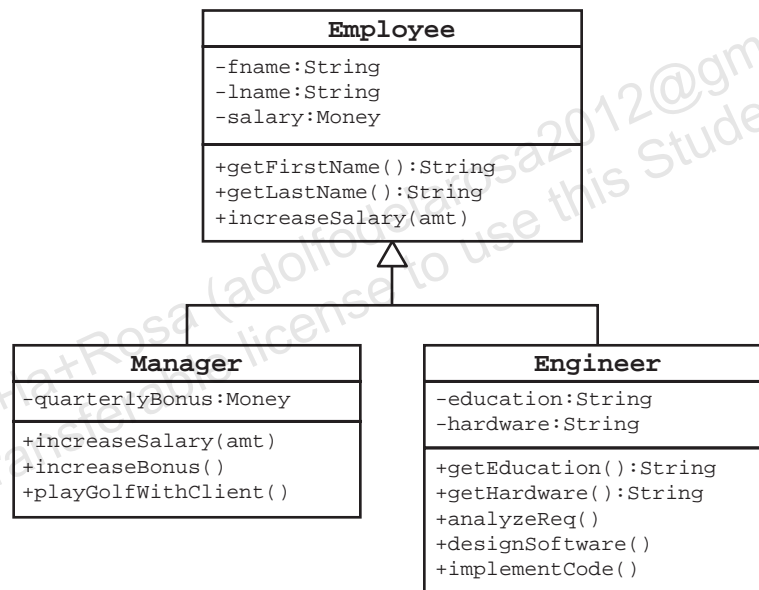
This principle was introduced by Barbara Liskov in 1987. A brief simplified interpretation is that if you substitute a subclass (for example, `Engineer`) for a superclass (for example, `Employee`), then any code that expects to use the superclass should have no surprises from a behavioral point of view. That

is, the code should find that all the methods it expected to find in the superclass are still provided and accessible, with any overridden methods having the same semantic meaning.

- A subclass can inherit from multiple superclasses (called multiple inheritance) *or* a subclass can *only* inherit from a single superclass (single inheritance).

Some OO languages support multiple inheritance, which enables a subclass to inherit members from multiple superclasses. C++ supports this feature. Unfortunately, this feature has some significant drawbacks when it comes to language design. The Java programming language only supports single inheritance.

Figure 1-7 illustrates an example inheritance hierarchy. The HR example has been extended with a new type of employee: Manager.



**Figure 1-7** An Example Inheritance Hierarchy

## Abstract Classes

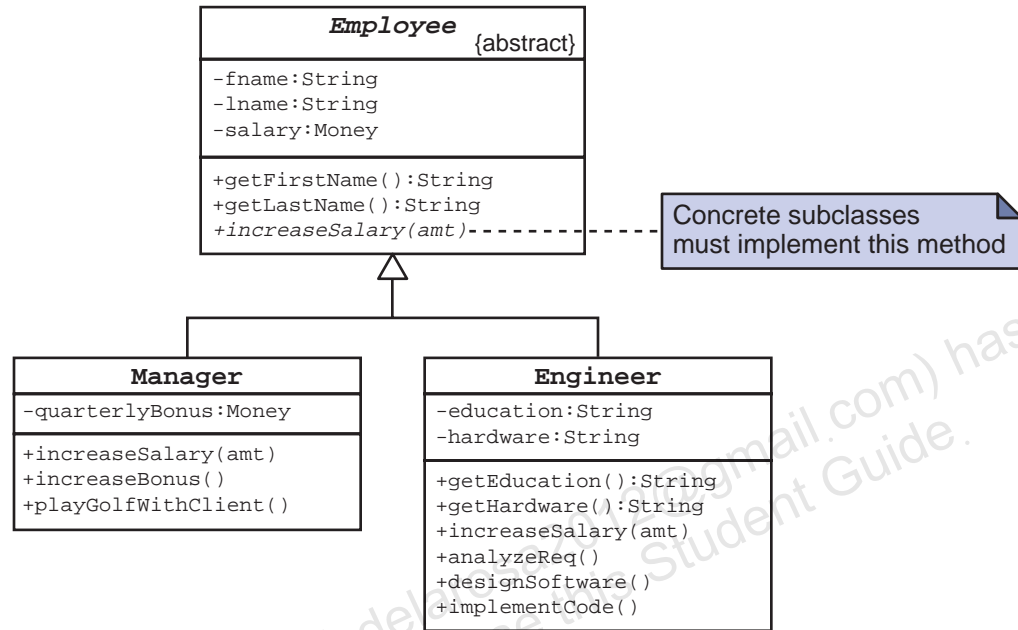
A class that contains one or more abstract methods, and therefore can never be instantiated. (Sun Glossary)

Features of abstract classes:

- Attributes are permitted.  
An abstract class may declare instance variables. These attributes are inherited by the subclasses of the abstract class.
- Methods are permitted and some might be declared abstract.  
Methods may be declared in an abstract class. One or more of the methods might also be declared abstract meaning that this class does not define an implementation of that method.
- Constructors are permitted, but no client may directly instantiate an abstract class.  
Constructors may be declared in an abstract class. These constructors can be used by the constructors of subclasses. However, the existence of constructors in an abstract class does not mean that the class may be instantiated. Instances cannot be created because one or more methods are abstract.
- Subclasses of abstract classes must provide implementations of all abstract methods; otherwise, the subclass must also be declared abstract.  
Subclasses of abstract classes may implement any of the abstract methods in its abstract superclass. The subclass may also override any non-abstract methods in the abstract superclass. However, if the subclass does not provide implementations for all abstract methods, then that subclass must also be declared abstract.
- In the UML, a method or a class is denoted as abstract by using italics, or by appending the method name or class name with {abstract}.  
It is difficult to use italics in handwritten drawings, which tend to use the {abstract} notation.

A *concrete class* is the opposite of an abstract class. A concrete class must have no abstract methods.

Figure 1-8 illustrates an example abstract class. The `Employee` class in the HR example has been declared abstract; likewise the `increaseSalary` method is abstract. Therefore, the `Manager` and `Engineer` classes must implement this method.



**Figure 1-8** An Example Abstract Class

## Interfaces

Features of Java technology interfaces:

- Attributes are not permitted (except constants).  
Interfaces by definition do not have attributes. However, interface may declare constants (with the modifiers `public static final`).
- Methods are permitted, but they must be abstract.  
Interfaces are a set of abstract methods. An interface may not declare a method with an implementation.
- Constructors are not permitted.  
Interfaces are not classes and no objects may be constructed from an interface. However, a class might implement an interface and objects of that class may be said to be *instances of the interface*. However, it is more appropriate to say the *instance supports the interface*.
- Subinterfaces may be defined, forming an inheritance hierarchy of interfaces.  
Interfaces may have subinterfaces. Therefore, a set of interfaces might form a hierarchy. For example, the Collections API in the `java.util` package includes a hierarchy of interfaces. The root interface is `Collection` with two subinterfaces `List` and `Set`; the `Set` interface also has subinterface `SortedSet`.

A class may implement one or more interfaces. This is sometimes called interface inheritance because the class inherits the methods of the interfaces. If the class does not implement these methods, then it must be declare abstract.

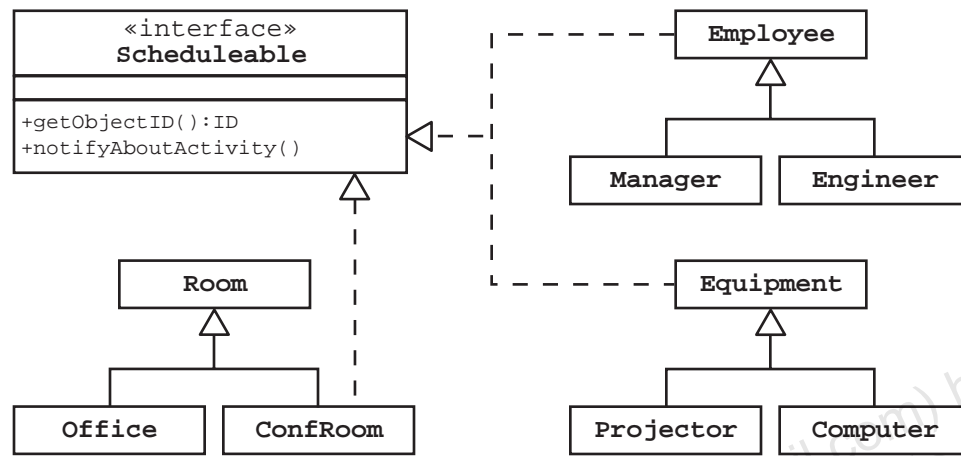



---

**Note** – The Java programming language supports interfaces directly. Other languages, such as C++, can support the concept of interfaces by defining classes that are completely abstract (all methods are virtual and empty) with no attributes.

---

Figure 1-9 illustrates an example interface. The HR example has been extended to include an interface for scheduling resources, such as people, rooms, and equipment.



**Figure 1-9** An Example Interface

**Note** – Interfaces can be implemented by any class in any hierarchy; classes that implement an interface are not bound to a single class hierarchy.

The **Employee** and **Equipment** classes implement the **Scheduleable** interface. Because these two classes are the root of a class hierarchy, any subclass of either of these superclasses are also “scheduleable.” The **Room** class hierarchy is different because the system should not be permitted to schedule activities for any arbitrary room, such as an office or a closet. However, conference rooms need to be scheduled; therefore, the **ConfRoom** class implements the **Scheduleable** interface independent of the other classes in the rooms hierarchy.

## Polymorphism

Polymorphism is “a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass [type].” (Booch OOAD page 517)

Aspects of polymorphism:

- A variable declared to reference type `T` can be assigned different types of objects at runtime provided they are a subtype of the variable's type `T`.

As shown in Figure 1-10 on page 1-26, the `employee` variable can reference any type of object whose class is a proper subclass of `Employee`. Polymorphism can be used with variables (local and instance), method parameters, and method return values.

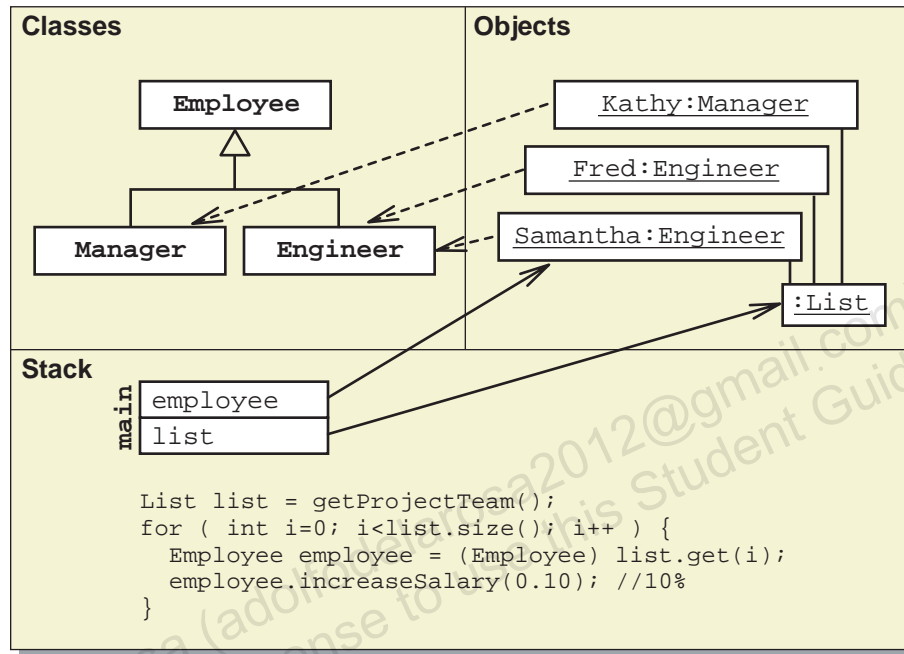
- Method implementation is determined by the type of object, not the type of the declaration.

Dynamic binding works by looking up the method (based on the signature) within the declared methods for the actual class of the object, not the apparent type (that is, the reference variable type). If the method is not found in that class, then the language environment performs the lookup on the superclass. If the method is not found in the superclass, then the method lookup is performed on the superclass's superclass, and so on.

- Only method signatures defined by the variable type can be called without casting.

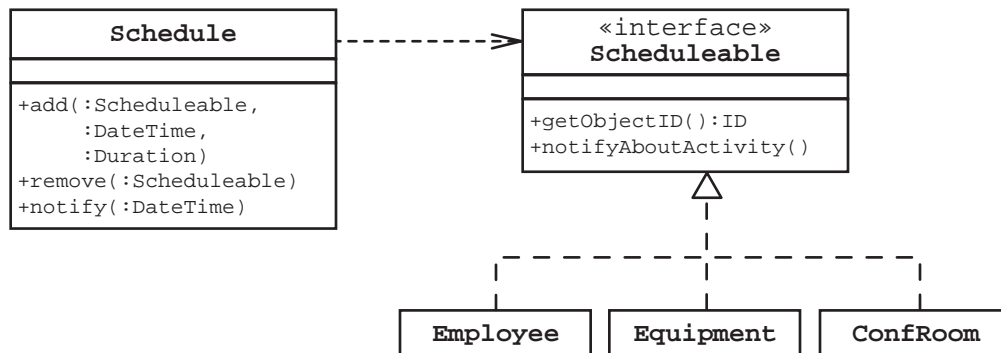
A method signature is the name of the method along with its parameters. A variable declared to reference type `T` can only be used to call methods that are supported by type `T`. Using the example shown in Figure 1-11 on page 1-26 any code using a variable declared to reference a `Scheduleable` type object can only call `getObjectID()` or `notifyAboutActivity()`.

Figure 1-10 illustrates an example of polymorphic behavior. The list contains three objects: one manager and two engineers. However, in the code, the employee variable is declared only as `Employee`; so how does the system know which `increaseSalary` method to call? The runtime environment finds the class of the actual object to determine which method to call. This language feature is called *dynamic binding*.



**Figure 1-10** An Example of Polymorphism

Figure 1-11 illustrates an example using interfaces. The `Schedule` class enables any kind of schedulable object to be added to a schedule. Therefore, employee objects, equipment objects, and conference room objects might be added to a schedule.



**Figure 1-11** A Polymorphism Example Using Interfaces



## Cohesion

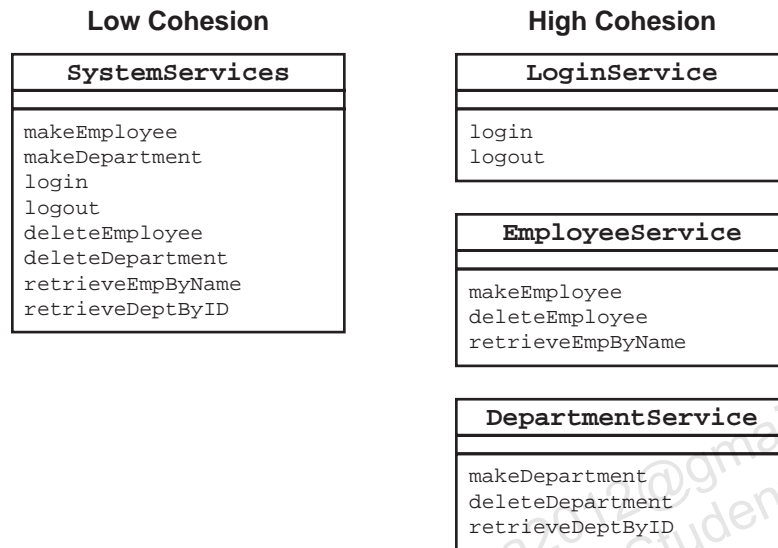
Cohesion is “the measure of how much an entity (component or class) supports a singular purpose within a system.” (Knoernschild page 174)

In software, the concept of cohesion refers to how well a given component or method supports a single purpose.

Aspects of cohesion:

- *Low cohesion* occurs when a component is responsible for many unrelated features.  
Big classes with many unrelated methods tend to be hard to maintain.
- *High cohesion* occurs when a component is responsible for only one set of related features.  
Small classes with fewer, but highly related, methods tend to be easier to maintain.
- A component includes one or more classes. Therefore, cohesion applies to a single class, a subsystem, and a system.
- Cohesion also applies to other aspects including methods and packages.
- Components that do everything are often described with the Anti-Pattern terms of Blob components  
Whereas Patterns (as covered later in the course) are generally desirable, Anti-Patterns are generally undesirable.

Figure 1-12 illustrates a class, `SystemService`, that has low cohesion. This class has functions that are completely unrelated: logging in and out, manipulating employees, and manipulating departments. The classes on the right side show a refactoring of the `SystemService` class into three highly cohesive classes.

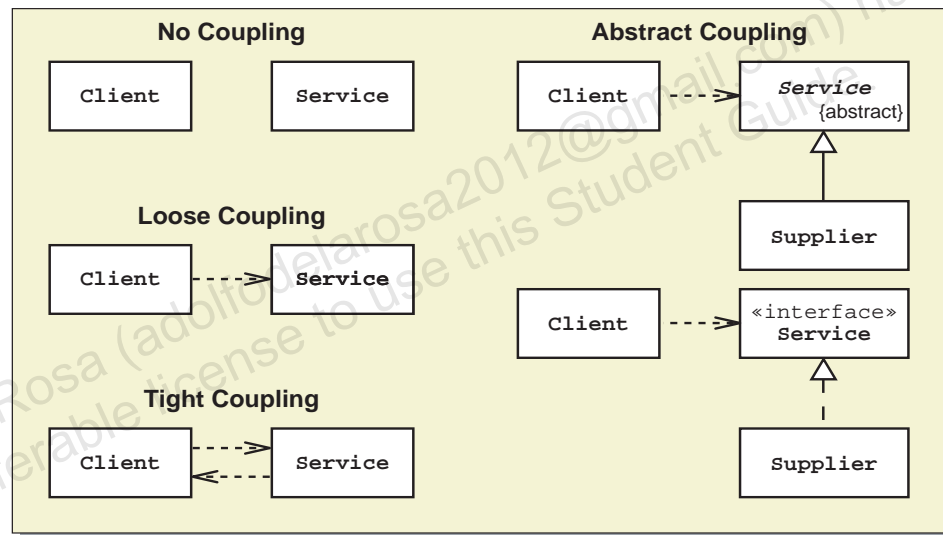


**Figure 1-12** An Example of Low and High Cohesion

## Coupling

Coupling is “the degree to which classes within our system are dependent on each other.” (Knoernschild page 174)

Coupling refers to the degree to which one class makes use of another class. The typical example is of a Client class that uses a Service class. In some cases a Client class never uses a Service class; this is called *no coupling*. A more interesting case is when the Client class that uses a concrete Service class; this is called loose coupling because the Client class has intimate knowledge of the Service class. Finally, tight coupling occurs when both the Client and Service class depend upon each other. The left side of Figure 1-13 shows these three cases.



**Figure 1-13** Four Forms of Coupling

The right side of Figure 1-13 shows two variations on a form of coupling called, *abstract coupling*. In this situation, the client requires the use of a service, but it does not know what concrete class is supplying that service. The Java programming language supports two abstract coupling mechanisms. One mechanism is to create an abstract Service class and then one or more concrete Supplier classes, which extend the Service class. Alternatively, you could create a Service interface and then one or more concrete Supplier classes, which implement the Service interface.

---

**Note** – Abstract coupling is an example of the Dependency Inversion Principle, which is covered later in the course.

---



## Class Associations and Object Links

### Class Associations

Class associations have many dimensions. These associations are discussed in greater detail later in the course. For now, you should know about these features:

- The roles that each class plays  
An association can be labeled with role names. These role names identify how each object is related to the other. For example, the association between `Department` and `Employee` as a role name of `dept` next to the `Department` class.
- The multiplicity of each role  
An association defines how many objects can participate in the association. For example, in the association between `Department` and `Employee`, an employee has only one (1) department, but a department can have one or more (1..\*) employees.

---

**Note** – Multiplicity of roles is covered in more detail later in the course.

---

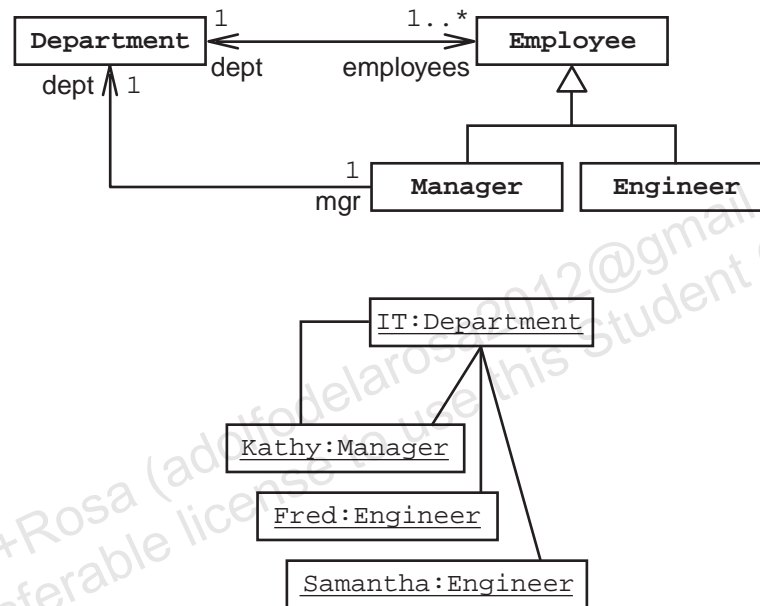
- The direction (or navigability) of the association  
An association can define the navigability of the association. This is the path leading from one object to another. For example, from the department object the system can navigate to any of the employees, and from an employee the system can navigate to that person's department. However, this structure specifies that the system can navigate from the manager to that manager's department; however, the department object cannot navigate to that department's manager.

### Object Links

- Object links are instances of the class association.  
Just as objects are instances of classes, object links are instances of the class association.
- Object links are one-to-one relationships.  
In a one-to-many or many-to-many class relationship, each link represents a connection from a single object to a single object.



Figure 1-14 illustrates the use of class associations and object links. The top diagram shows a set of classes and their associations. The `Department` class is associated with one or more objects of the `Employee` class. The `Manager` class is associated with one (and only one) object of the `Department` class. The bottom diagram shows one example of a group of objects and their runtime links at a particular moment in time. The IT department contains three employees, and Kathy is the manager of the IT department. If we add another employee (or subtype) to the department, then the bottom (object) diagram would change, provided that the multiplicity constraints of the top (class) diagram allow the change.



**Figure 1-14** An Example of Object Associations

## Delegation

Many computing problems can be easily solved by delegation to a more cohesive component (one or more classes) or method.

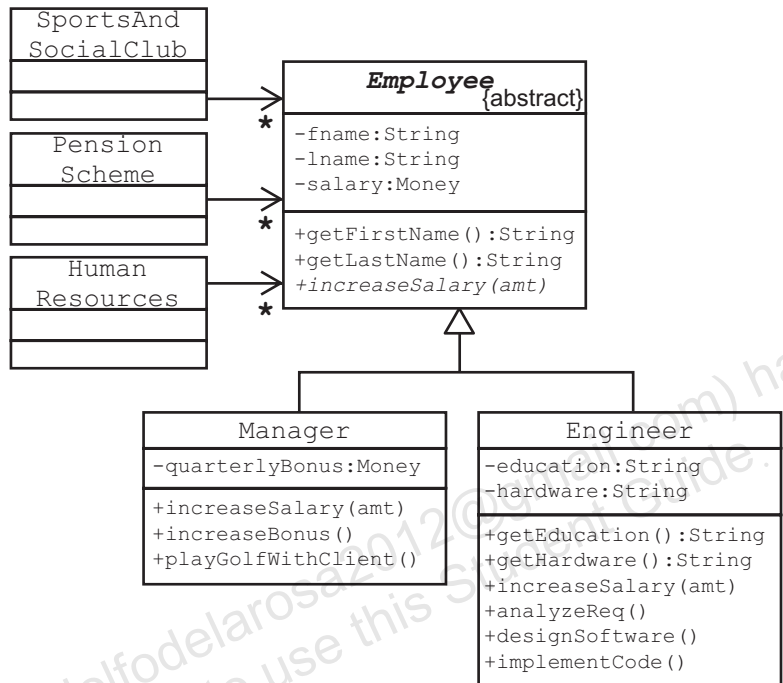
- Delegation is similar to how we humans behave. For example:
  - A manager often delegates tasks to an employee with the appropriate skills.
  - You often delegate plumbing problems to a plumber.
  - A car delegates accelerate, brake, and steer messages to its subcomponents, who in turn delegate messages to their subcomponents. This delegation of messages eventually affects the engine, brakes, and wheel direction respectively.
- OO paradigm frequently mimics the real world.
- The ways you delegate in OO paradigm include delegating to:
  - A more cohesive linked object
  - A collection of cohesive linked objects
  - A method in a subclass
  - A method in a superclass
  - A method in the same class

Figure 1-15 on page 1-33 illustrates one of the many problems that delegation can solve. The figure shows the example from Figure 1-8 on page 1-22 with some additional classes added. These are just a few of the additional classes whose objects might have links to the Employee objects at some time. In this business problem, it is likely that employees change their job roles. For example, an engineer might be promoted to a manager. However, you cannot change your class. So, for an engineer to be promoted to a manager, you would have to do the following:

1. Create a new object of type Manager and copy the common attribute values (fname, lname, and salary) to the new object.
2. Ask all the objects that use the Engineer object to replace that link with a link to the Manager object.
3. Delete the Engineer object.

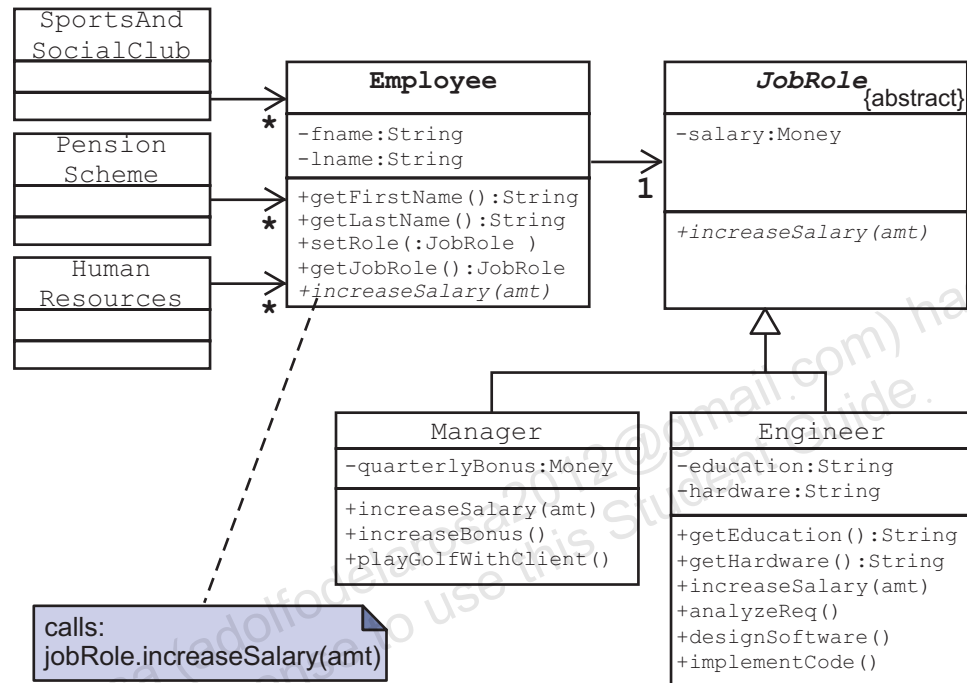


**Note** – We have not addressed the issues of keeping a historical record of an employee’s employment.



**Figure 1-15** Issues with Inheritance on Non-Coherent Classes

The problem in this example is because the details of job roles and non-job roles are mixed together, which is not coherent. Figure 1-16 shows one possible solution that separates these two aspects and associates the two classes. In this case, the `Employee` will delegate the job role to the `JobRole` class that is subclassed into `Engineer` or `Manager`.



**Figure 1-16** Delegation to a more Cohesive Class

**Note** – As this is an introduction, we have not covered all of the aspects of delegation. For example, we have not covered having multiple delegates, such as roles or types. These aspects will be covered later in the course.

**Note** – Delegation is not only for roles; it has many other discriminators—for example, types.



# Summary

Object orientation is a model of computation that is closer to how humans think about problems. Using an OO paradigm for developing systems has many benefits over procedural development because of the closer mapping between your mental models (the system requirements), your analysis and design models, and your implementation.

OO provides a set of fundamental concepts:

- Object
- Class
- Abstraction
- Encapsulation and information hiding
- Inheritance
- Abstract classes
- Interfaces
- Polymorphism
- Cohesion
- Coupling
- Class associations and object links
- Delegation

These OO concepts are the basis of most OO principles and patterns used throughout this course.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Introducing Modeling and the Software Development Process

---

## Objectives

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

- Describe the Object-Oriented Software Development (OOSD) process
- Describe how modeling supports the OOSD process
- Describe the benefits of modeling software
- Explain the purpose, activities, and artifacts of the following OOSD workflows (disciplines): Requirements Gathering, Requirements Analysis, Architecture, Design, Implementation, Testing, and Deployment

## Additional Resources



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

- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Harlow: Addison Wesley Longman, Inc., 1993
- Larman, Craig. *Applying UML and Patterns (3rd ed)*. Upper Saddle River: Prentice Hall, 2005.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004

# Exploring the OOSD Process

The software industry has experienced exponential growth and change in the half-century from its birth. There has been a great deal of change in the technologies of programming: languages, operating systems, networking, communication protocols, component-based Application Programming Interfaces (APIs), and application server software. The software development process has changed along with the technologies. When using Object-Oriented (OO) technologies for development, then you should use OOSD processes because the OO technologies influence the software development processes.

There have also been many changes about how software projects are organized and managed. Who determines what the software is required to do? How is the software solution created? How is the development life cycle managed? The answers to these questions guide the OOSD process.



---

**Note** – There is no “one standard OOSD process.” The concepts and activities presented in the majority of this course demonstrates a generic (vanilla) approach. The goal of this course is to provide you with techniques that you can use in any process (methodology). Later in the course, you will be introduced to some of the specific OO development processes that are available.

---

## Describing Software Methodology

In Webster's New Collegiate Dictionary, *methodology* is defined as "a body of methods, rules, and postulates employed by a discipline." In software development, a methodology refers to the highest-level of organization of the development process itself.



---

**Note** – Technically, the term methodology means "the science which studies methods" and the term method means "the composition of a language and a process." Therefore, it is more appropriate to say that the Unified Process (UP) is a method, not a methodology. However, it is common practice to refer to UP (and other OOSD methods) as a methodology. This is the terminology used in this course.

---

Over the history of the software industry, there have been hundreds of methodologies developed. OO methodologies incorporate object-oriented concepts throughout the OOSD process. Many modern OO methodologies compose the development process into large-scale *phases*, such as Inception, Elaboration, Construction, and Transition. These phases are composed of *workflows (disciplines)* and these workflows are composed of specific *activities*. Activities involve workers and artifacts. A *worker* is a person that performs the activity. An *artifact* is a tangible piece of information that is produced by an activity. Artifacts such as diagrams, documents, and the software code itself are produced with tools. The Unified Modeling Language (UML) is one of our most powerful tools for modeling software.

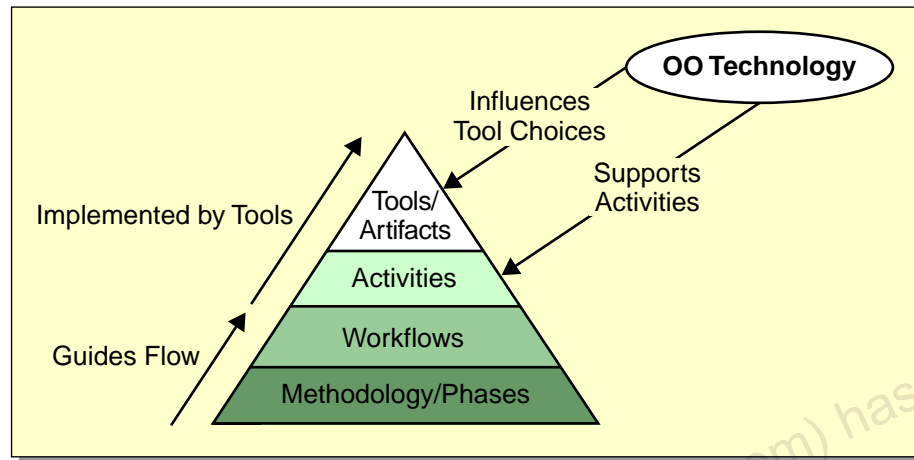


---

**Note** – The course uses the term *workflow*. However, the Object Management Group (OMG) is recommending a new term *discipline* to replace workflow.

---

Figure 2-1 illustrates the hierarchical relationships between methodology phases, workflows, activities, and the tools that create artifacts.



**Figure 2-1** OOSD Hierarchy Pyramid<sup>1</sup>

To support the activities, the development team uses various tools to analyze, model, and construct the software solution. These tools include word processors, the UML, UML modeling tools, advanced text editors, and integrated development environments.

An artifact produced in one activity might be an input into another activity. For example, a Use Case diagram (which describes the intended behavior of the system) is used during the Design workflow to determine the software components needed to satisfy the functional requirements defined by the use cases.

Artifacts are meant to be long-lived, but several activities often produce short-lived modeling artifacts that are discarded because the information gained is either documented elsewhere more formerly or is found to be of no value.

Artifacts can be documents, diagrams, and even a functioning system. The goal of software development is to produce a functioning system (the final artifact) that satisfies the requirements of the business owner, users, and other client-side stakeholders.

1. This OOSD hierarchy pyramid was influenced by Jacobson's pyramid of a "rational enterprise philosophy." The hierarchy of Jacobson's pyramid is slightly different than the one shown in Figure 2-1. It has the following structure: Architecture at the bottom, then Method, then Process, and then Tools at the top. Read *Object-Oriented Software Engineering* (section 1.2) for more information.

## Listing the Workflows of the OOSD Process

Workflows consist of activities performed by workers on the development team. There are many possible workflows, but this course focuses on the following seven:

- *Requirements Gathering*  
Determine the requirements of the system by meeting the business owner and users of the proposed system.
- *Requirements Analysis* (or just Analysis)  
Analyze, refine, and model the requirements of the system.
- *Architecture*  
Identify risk in the project and mitigate the risk by modeling the high-level structure of the system.
- *Design*  
Create a Solution model of the system that satisfies the system requirements.
- *Implementation*  
Build the software components defined in the Solution model.
- *Testing*  
Test the implementation against the expectations defined in the requirements.
- *Deployment*  
Deploy the implementation into the production environment.

There is a single common goal that binds all of these workflows together:  
*To deploy a tested software system that satisfies all of the requirements defined by the business owner and users.*

---

**Note** – Some people include *maintenance* in the list of workflows. This course considers maintenance as a new revision of the software and each revision embodies the whole development process.

---

---

**Note** – Some companies use different terminology for these workflows (disciplines). Implementation is sometimes called build, and deployment is sometimes called implementation.

---

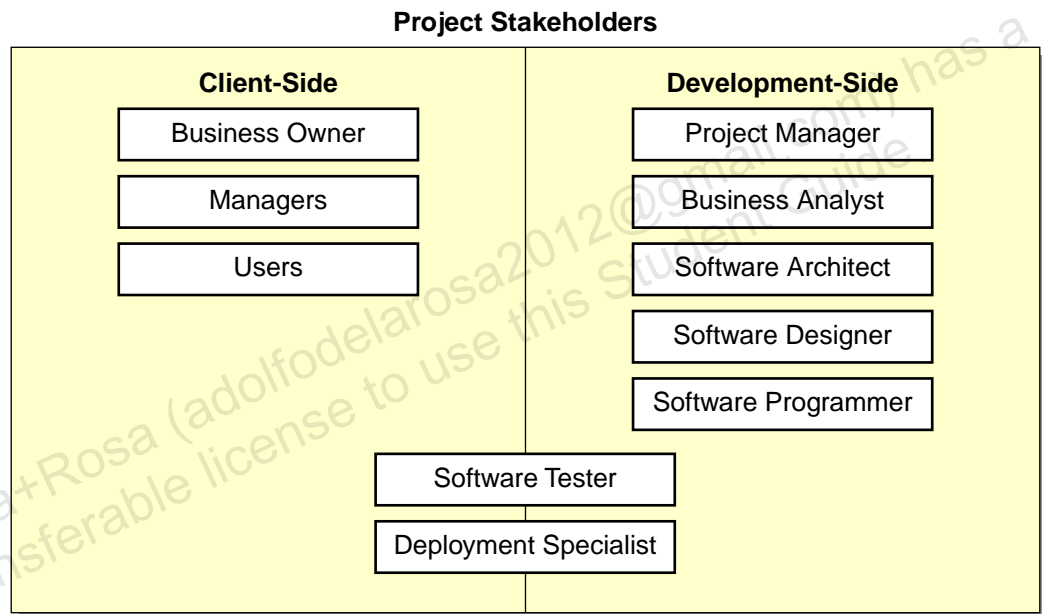




## Describing the Software Team Job Roles

A stakeholder is any person or group that has an interest in the project. The set of stakeholders includes users and managers on the client-side and the complete project team on the development side.

As previously mentioned, workers perform activities which generate artifacts that document, model, and implement the software solution. There are many job roles on a software project; however, some workers might fulfill multiple job roles. Figure 2-2 shows a common set of OOSD job roles.



**Figure 2-2** Software Development Job Roles

- *Business Owner*  
The lead stakeholder on the client-side of the project. This person is responsible for making final decisions about the behavior of the system.
- *Users*  
Any person that will be using the proposed system. For internal systems, these are usually employees of the client company. For Web-based systems, this could be any Internet user.
- *Managers*  
The people that are the managers of internal users.
- *Project Manager*

Manages the software development project. This is usually *not* a technical manager. They manage the budget, resources, and schedule of the project. They are often the account manager for the client (if consulting).

- *Business Analyst*

Gathers requirements from the client-side stakeholders and analyzes the functional requirements by modeling the *enduring business themes* of the system. This role is responsible for the Business Domain diagrams, which include Use Case diagrams and Business Domain (Analysis) class diagrams.

*Synonyms:* Enterprise Modeler, Domain Modeler, and System Analyst

- *Software Architect*

Defines the architecture of the system, leads the development of the architectural baseline during the Inception and Elaboration phases, analyses the NFRs, identifies project risk, and creates a risk mitigation plan.

- *Software Designer*

Creates the Solution model of the system based on the Business Domain diagrams within the framework of the architecture.

*Synonyms:* Software Engineer

- *Software Programmer*

Implements the software solution. In many small development teams, the programmer and designer roles are filled by the same person.

*Synonyms:* Software Developer

- *Software Tester*

Tests the implementation to verify that the system meets the requirements (both FRs and NFRs). This role might include the development team for unit and integration testing, as well as the client-side Quality Assurance (QA) personnel for acceptance testing.

*Synonyms:* QA Specialist, Test Engineer

- *Deployment Specialist*

Deploys the implementation onto the production platform. This role might include several other job roles: System Administrator, Network Administrator, Script Writer, and so on. This role is performed by the development team during construction, but when the system goes to production this role should be filled by the client organization.

# Exploring the Requirements Gathering Workflow

The requirements of a system are divided into two fundamental categories: functional requirements (FRs) and non-functional requirements (NFRs). The FRs describe the behavior of the system relative to the perspective of the actors that use the system. High-level FRs are visualized as *use cases*.



**Note** – This course will first develop use cases (high-level FRs) and then drill down to determine the lower-level behavior for each use case, which implicitly includes the lower-level FRs.

The NFRs describe the quality of service of the system. These requirements include such characteristics as performance (measured in response time), throughput (measured by how many simultaneous users can be accommodated), and so on.

## Workflow Purpose and Job Roles

The purpose of the Requirements Gathering workflow is to determine the requirements of the system by meeting the business owner and users of the proposed system. Table 2-1 shows the description of this workflow.

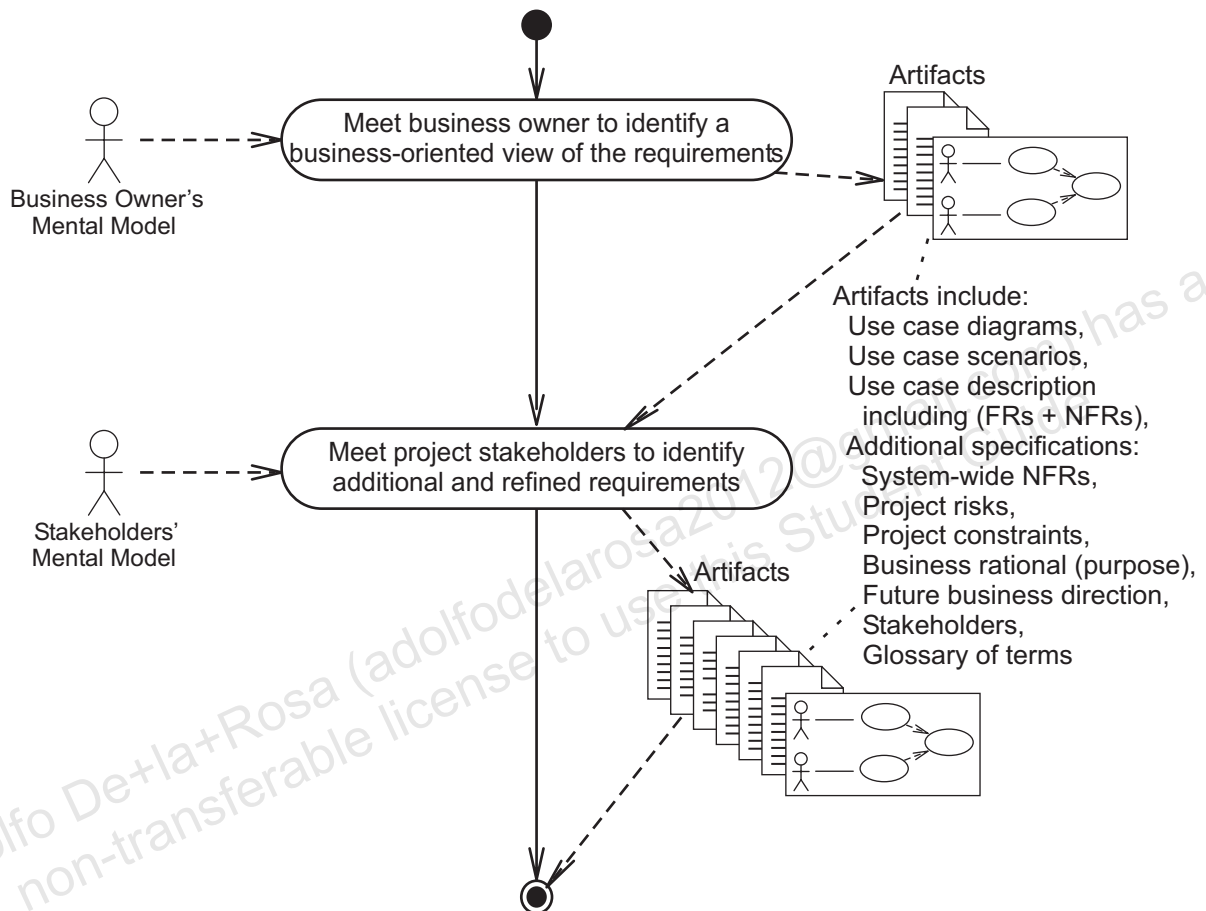
**Table 2-1** Requirements Gathering Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	Determine: <ul style="list-style-type: none"> <li>• With whom the system interacts (actor)</li> <li>• What behaviors (called use cases) that the system must support</li> <li>• Detailed behavior of each use case, which includes the low-level FRs</li> <li>• Non-functional requirements</li> </ul>

The Requirements Gathering workflow activities are usually performed by business analyst and the architect job roles.

## Workflow Activities and Artifacts

Figure 2-3 illustrates the activities the Requirements Gathering workflow might include.



**Figure 2-3** Activities and Artifacts of the Requirements Gathering Workflow

The Requirements Gathering activities include:

- Meeting with the business owner to gather the business-significant use cases and some business-specific low-level details

To determine the size and scope of a software project, the architect and business analyst meet with the business owner to determine the high-level FRs of the system in the form of use cases. The output of this activity are artifacts that describe a business-oriented vision of the system. These artifacts include:

- Use case diagrams
- Use case scenarios

- Use case description (which include FRs and NFRs)



**Note** – The detailed flow of events in the use case description may be partially completed at this stage. However, it will be completed in the Analysis workflow.

- Additional specification document(s), which include:
  - Additional system-wide NFRs
  - Project risks
  - Project constraints
  - Business rational (purpose)
  - Future directions of the business
  - Client-side stakeholders
  - Glossary of terms
- Meet with the other project stakeholders to gather additional information and elaborate and verify previous information. The artifacts are the same as above but each stakeholder input will add or refine the previous artifacts.



**Note** – There are many variations to this workflow and the created artifacts. Some of these variations will be discussed later in the course. However, two significant variations are briefly mentioned below.

## Significant variations to the Requirements Gathering Workflow

Two significant variations to the Requirements Gathering workflow are as follows:

- In an iterative and incremental development process, you will add information and refinements in each iteration.
- In a non-incremental process (for example, Waterfall), you will often create a Vision document (or similar document) containing the requirements gathered from the business owner, and a System Requirement Specification document (or similar document) containing all the requirements gathered from all stakeholders.

## Exploring the Requirements Analysis Workflow

There are two views of the Requirements model created in this workflow:

- The completed *Use Case Forms*, containing full details of the actor's interaction with the system and what the system does in response.
- The *Domain model* (a Class diagram of the *key abstractions* of the problem space).

These artifacts and the previously created artifacts together make up the *problem space*. The language used in these artifacts should be understandable to the client-side stakeholders because they use this language to discuss their business (also known as a *domain*). For example, in a Hotel Reservation System the problem space uses terms like customer, rooms, hotel properties, payment methods, charge items, and so on. These terms are often the key abstractions of the problem space and they are modeled with a Domain model.

### Workflow Purpose and Job Roles

The purpose of Requirements Analysis workflow is to analyze, refine, and model the requirements of the system. Table 2-2 shows the description of this workflow.

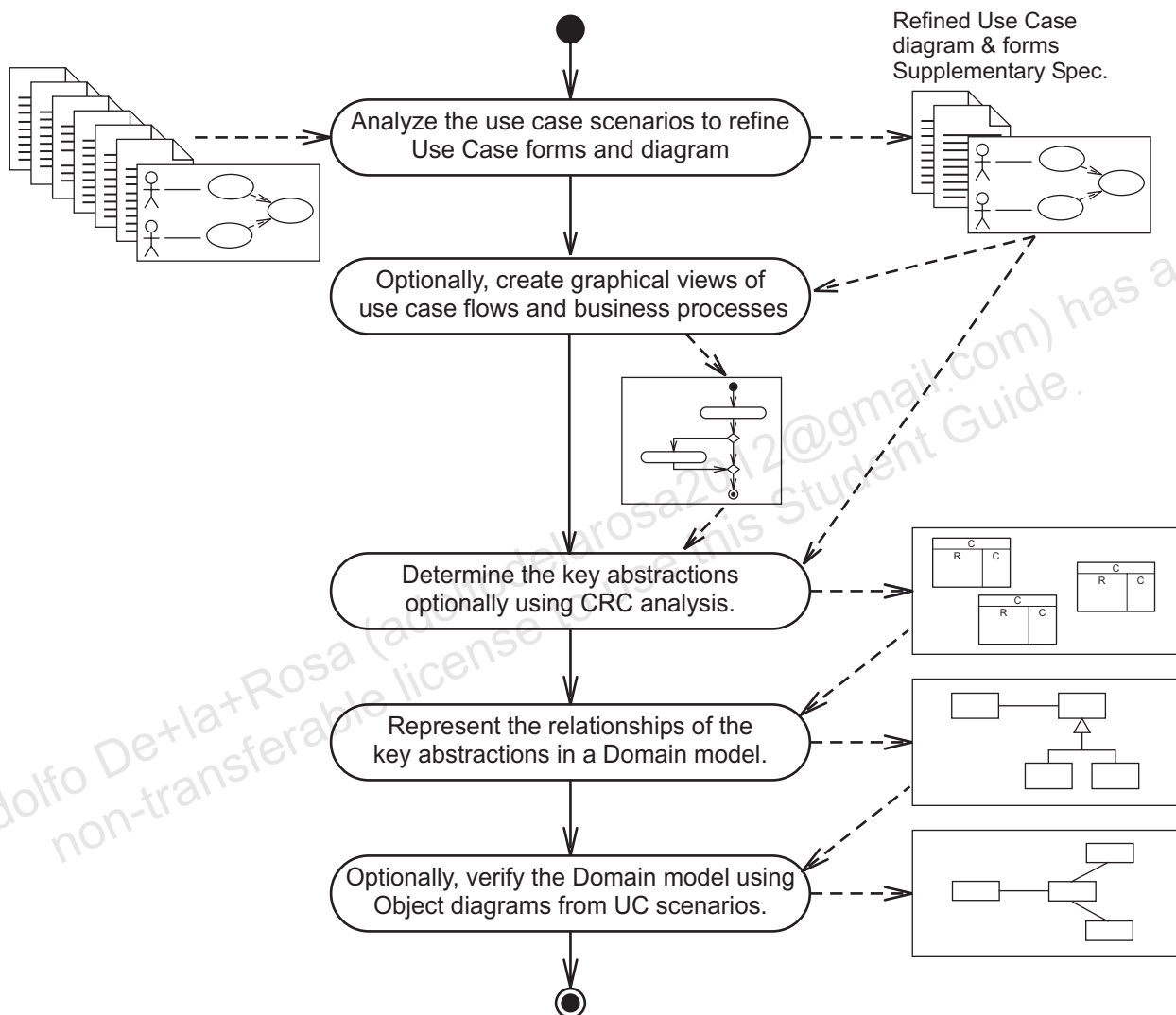
**Table 2-2** Requirements Analysis Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	
Requirements Analysis	Model the existing business processes	Determine: <ul style="list-style-type: none"> <li>• The detailed behavior of each use case</li> <li>• Supplementary use cases</li> <li>• The key abstractions that exist in the current increment of the problem domain</li> <li>• A business domain class diagram</li> </ul>

The Requirements Analysis workflow activities are usually performed by business analyst and the architect job roles.

## Workflow Activities and Artifacts

Figure 2-4 illustrates the activities the Requirements Analysis workflow might include.



**Figure 2-4** Activities and Artifacts of the Requirements Analysis Workflow

The Requirements Analysis activities are:

- Analyze the use case scenarios to discover more detail.  
Use scenarios to refine the Use Case forms. Add or refine the main flow and alternative flows to this form. Also, analyzing use case scenarios often identifies common patterns in use cases. These

common patterns can be made explicit in the Use Case form. For example, a common pattern of many use cases might be the need to log on to the system.

- Refine the Use Case diagram from the analysis.

For example, the log-on behavior can be captured in a separate use case node and have other use cases refer to the log-on use case.

- Optionally, create an Activity diagram that provides a visual view of the flow of events to augment and verify the textual Use Case forms.

A graphical view of the flow of events can be used to augment and verify the stakeholders understanding of the problem. Use a UML Activity diagram to model the activities of a use case at a fine level of granularity. These Activity diagrams can be shown to UML-savvy stakeholders to verify the analyst's understanding of the necessary behavior of the system. In their simplest form, Activity diagrams have some similarity to flow charts. Therefore, any stakeholders who understand flow charts should understand these diagrams.

Activity diagrams have many other uses, but at this point the only other use worth discussing is to use them in modeling the business process, which may include multiple use cases and activities that are conducted external to the system. There are several different non-UML formal Business Processing Modeling Notations (BPMN), which may be more suited for this purpose.

- Determine the key abstractions. This might be done by using CRC analysis, however there are other techniques to determine the key abstractions.

CRC analysis is one technique for identifying the key abstractions of the problem domain. CRC stands for Class Responsibility Collaboration.

- Represent the relationships of the key abstractions in a Domain model.

After the key abstractions have been identified, a UML Class diagram is created or appended to represent the key abstractions discovered and their interrelationships.

- Optionally, verify the Domain model using Object diagrams from use case scenarios.

The entities and data in a use case scenario can be modeled in Object diagrams. Verify the associations and multiplicities on the Domain model by checking the object links in the Object diagrams.



## Exploring the Architecture Workflow

The Architecture workflow is rather complex and is beyond the scope of this course. However, you will see how the Architecture model (developed in this workflow) affects the Design workflow. From the perspective of the designer, the Architecture model provides a template of the high-level system structure into which the designer plugs in designed components.

### Workflow Purpose and Job Roles

The purpose of the Architecture workflow is to identify risk in the project and to mitigate the risk by modeling the high-level structure of the system. Table 2-3 shows the description of this workflow.

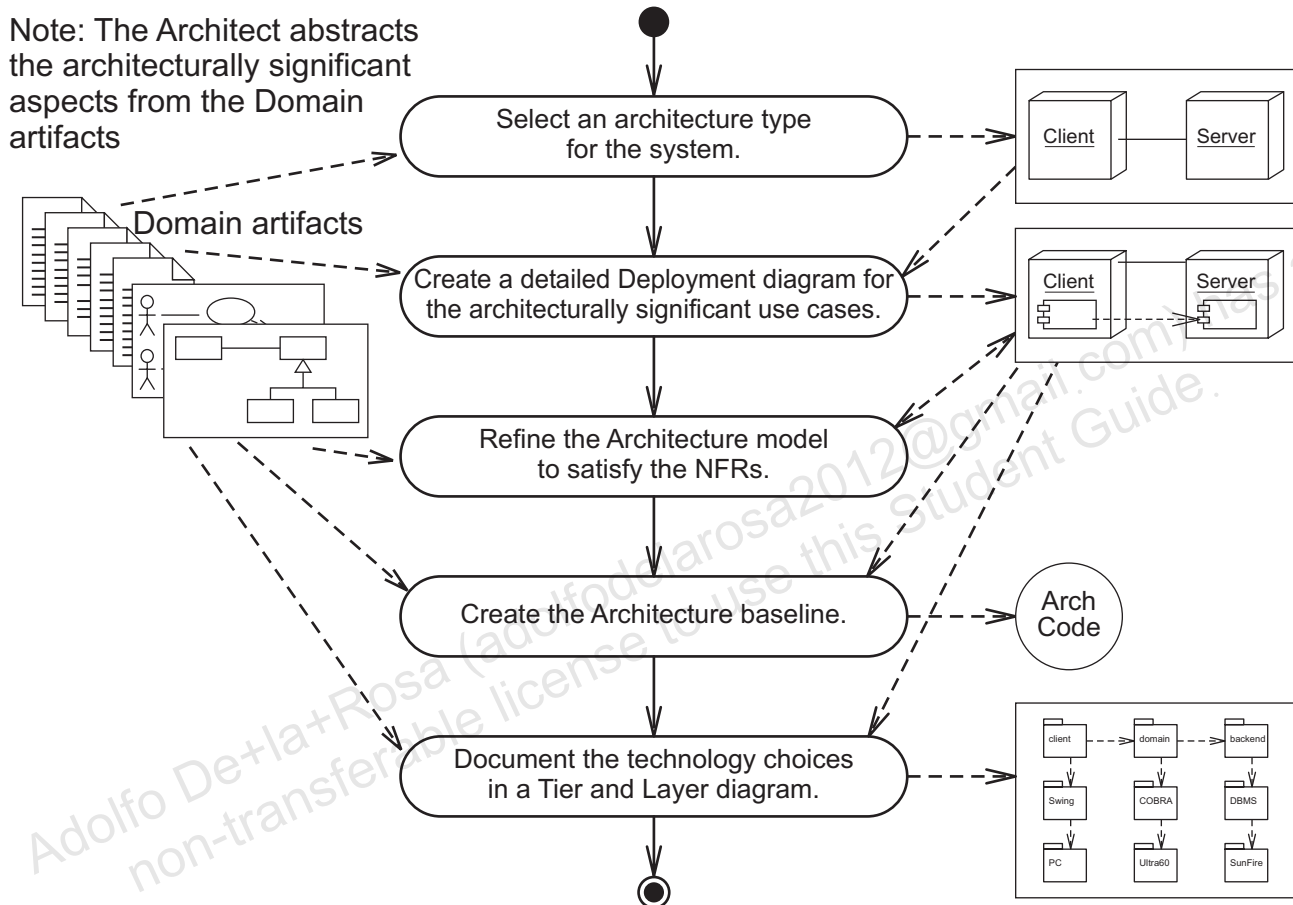
**Table 2-3** Architecture Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	<ul style="list-style-type: none"> <li>• Develop the highest-level structure of the software solution</li> <li>• Identify the technologies that will support the Architecture model</li> <li>• Elaborate the Architecture model with Architectural patterns to satisfy NFRs</li> </ul>

The Architecture workflow activities are performed by the architect.

## Workflow Activities and Artifacts

Figure 2-5 illustrates the Architecture workflow usually includes six activities.



**Figure 2-5** Activities and Artifacts of the Architecture Workflow

The Architecture activities are:

- Select an architecture type for the system.  
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.  
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.
- 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. This diagram shows the low-level components and their dependencies with the high-level Deployment diagram.

- 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 discussion is beyond the scope of this course.

- 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.

- Document the technology choices in a Tiers and Layers diagram.  
For each tier, and each layer, the architect identifies the appropriate technologies to be used.

## Exploring the Design Workflow

The ultimate goal of the Design workflow is to develop a Solution model that the development team can use to construct the code for the proposed system.

### Workflow Purpose and Job Roles

The purpose of the Design workflow is to create a Solution model of the system that satisfies the behavior of the system as defined by the requirements. Table 2-4 shows the description of this workflow.

**Table 2-4** Design Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	
Design	Model <i>how</i> the system will support the use cases	<ul style="list-style-type: none"> <li>• Create a Design model for a use case using Interaction diagrams</li> <li>• Identify and model objects with non-trivial states using a State Machine diagram.</li> <li>• Apply design patterns to the Design model</li> <li>• Create a Solution model by merging the Design and Architecture models</li> <li>• Refine the Domain model</li> </ul>

The Design model for the use cases is based on the classes discovered during analysis, which are the business objects. By using Interaction diagrams and State Machine diagrams, you can explore the detailed object interactions and state transitions. Doing this will enable you to discover new support classes as well as missing attributes and methods for the domain classes.

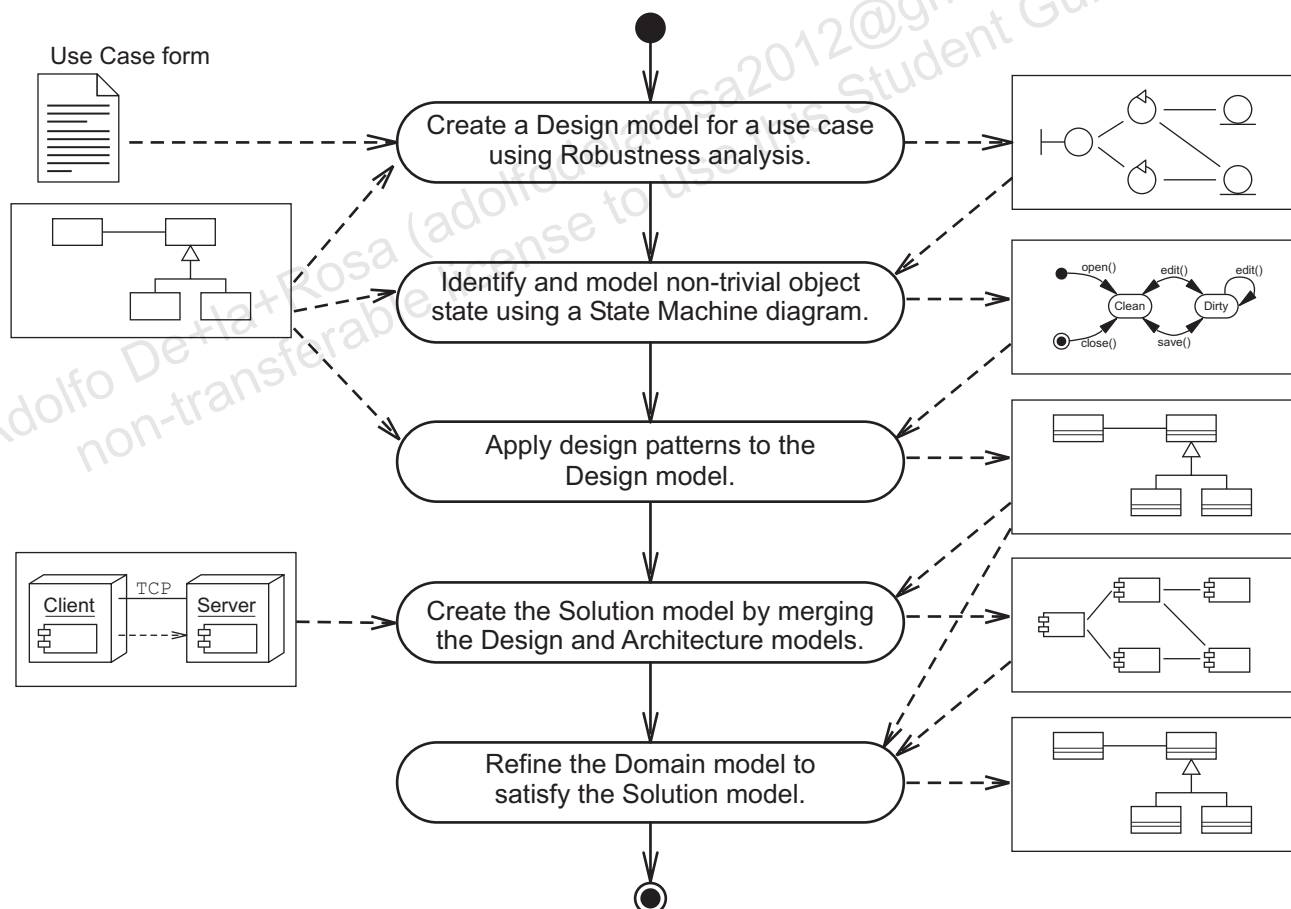
The Design model is then merged with the Architecture model. This combined model is called the Solution model, in this course. The Solution model also includes the refined Domain model.

The Solution model is also refined by a variety of Design patterns that are applicable to the design problem and context. The components in the Solution model can be implemented in code.

The Design workflow activities are performed by the software designer job role.

## Workflow Activities and Artifacts

Figure 2-6 illustrates the Design workflow usually includes six activities.



**Figure 2-6** Activities and Artifacts of the Design Workflow

The Design activities are briefly described here:

- Create a Design model for a use case using Interaction (or Robustness analysis) diagrams.

Using Interaction diagrams, you add new classes to enable to Domain objects to work with the computer system. By following the flows in a use case (and its scenarios), you can discover a collection of collaborating software components that satisfy the functional requirements of the use case. These usually include Service classes and Boundary classes. During this process you might discover missing methods and attributes in your Domain classes.

- Identify and model objects with non-trivial states using a State Machine diagram.

A small, but significant number of classes have non-trivial state dependencies, where their behavior to any event depends on their current state. For these classes, you can model the states, trigger events, conditions, and actions that can occur. State Machine diagrams are often used in combination with Interaction diagrams to check that every scenario that could occur has been considered. For example, an automatic gearbox in a car should react differently when you select reverse, depending on whether you are stationary or traveling forward at a speed of 70 mph.

- Apply design patterns to the Design model.

There are now hundreds of design patterns that have been documented. This activity enables the designer to refine the Design model with applicable patterns to make the software more flexible and robust. These patterns can be added at any time during design.

- Create the Solution model by merging the Design and Architecture models.

In this activity, the designer inserts the components from the Design model into the Architectural model. This structure provides about 80 percent of the components that must be coded during the Implementation workflow.

- Refine the Domain model to satisfy the Solution model.

The Domain model from the Analysis workflow must be refined to provide enough details (attributes and methods) for the Implementation workflow.

# Exploring the Implementation, Testing, and Deployment Workflows

These three workflows are outside the scope of this course. However, they will be covered in the appendix so that you can see how these other workflows complete the software development process.

## Workflow Purpose and Job Roles

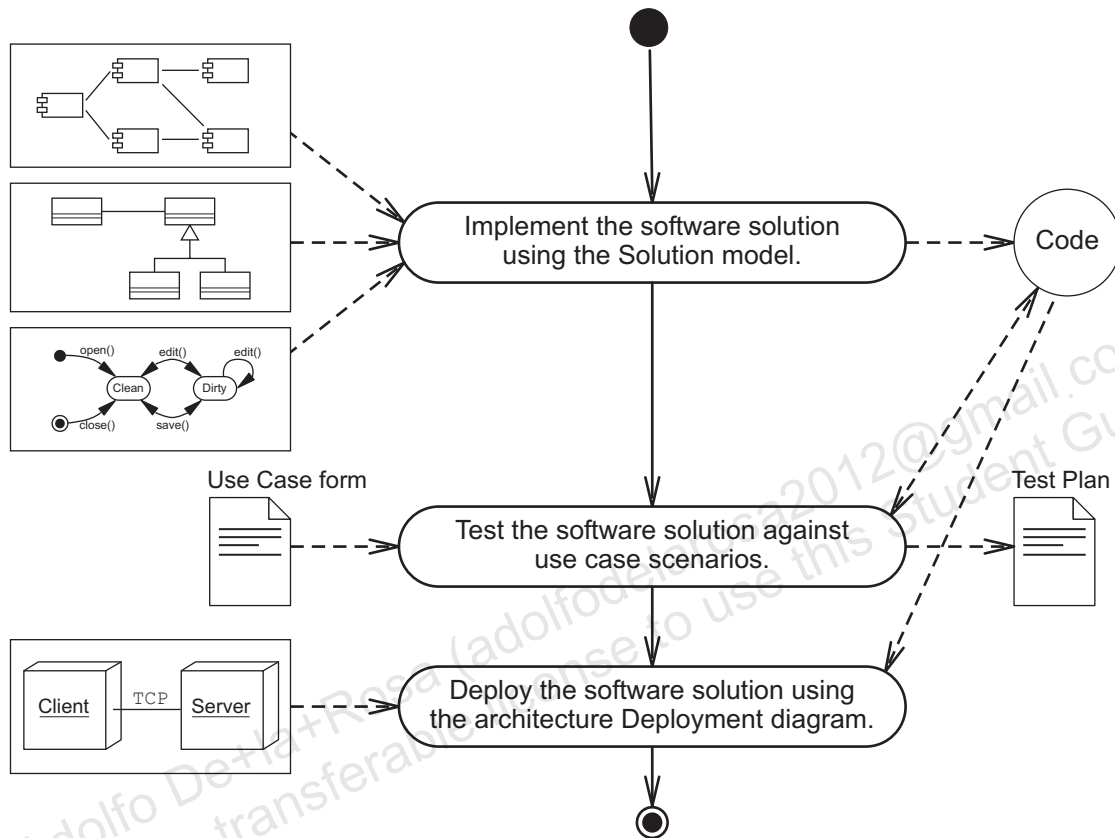
The purpose of the Implementation workflow is to build the software components defined in the Solution model. The Implementation workflow is performed by the software programmer job role. The purpose of the Testing workflow is to test the implementation against the expectations defined in the requirements. The Testing workflow is performed by the software tester job role. The purpose of the Deployment workflow is to deploy the implementation into the production environment. The Deployment workflow is performed by the deployment specialist job role. Table 2-5 shows the description of this workflow.

**Table 2-5** Implementation, Testing, and Deployment Workflows

Workflow	Purpose	Description
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	
Design	Model how the system will support the use cases	
Implementation, Testing, and Deployment	Implement, test, and deploy the system	<ul style="list-style-type: none"> <li>• Implement the software</li> <li>• Perform testing</li> <li>• Deploy the software to the production environment</li> </ul>

## Workflow Activities and Artifacts

Figure 2-7 illustrates the following activities that the Implementation, Testing, and Deployment workflows include.



**Figure 2-7** The Activities and Artifacts of the Implementation, Testing, and Deployment Workflows

The Implementation activity is: Implement the software solution using the Solution model. This activity maps the class structure defined in the refined Domain model into a physical, Java technology class structure.

The Testing activities are numerous and varied. This course focuses on acceptance testing. This activity is: Test the software solution against the use case scenarios. In this activity, the tester generates the Functional test plan from the use case scenarios. The tests are performed to verify that the functional behavior of the system matches the use case requirements.

The Deployment activity is: Deploy the software solution using the architecture Deployment diagram. In this activity, the deployment specialist uses the Deployment diagram to set up the computer, network, and component structure of the production environment.



# Examining the Benefits of Modeling Software

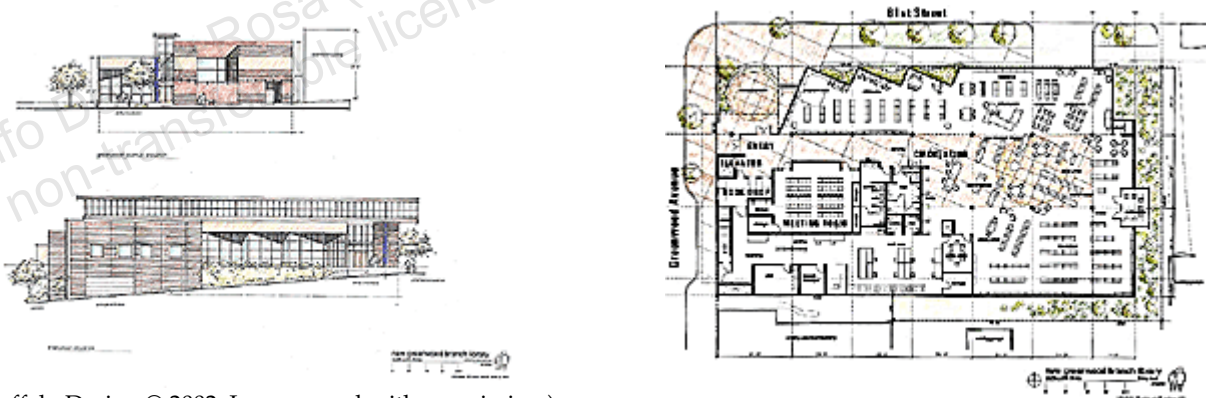
The inception of every software project starts as an idea in someone's mind. To construct a realization of that idea, the development team must create a series of conceptual models that transform the idea into a production system.

## What Is a Model?

"A model is a simplification of reality." (Booch UML User Guide page 6)

*"A description of static and/or dynamic characteristics of a subject area, portrayed through a number of views (usually diagrammatic or textual)."* (Larman, page 617)

You can think of a model in terms of traditional architecture. Before constructing a building, an architect draws several different views of the building itself and the location of the building. Figure 2-8 shows two diagrams from the same architectural plans for a library in Seattle. These different diagrams show different views.



(Buffalo Design © 2002. Images used with permission.)

**Figure 2-8** Example Models From Traditional Architecture

From architecture you can learn that:

- A model is an abstract conceptualization of some entity (such as a building) or a system (such as software).

A model is abstract by its nature. Models can represent many different things, both physical such as buildings and computer networks, and conceptual such as software.

- Different views show the model from different perspectives.  
An architect draws many diagrams showing different views or perspectives of the building. Views include: floor plans, plumbing, electrical wiring, external construction, street plans, and so on. A software model can also be represented by different views, and you can create diagrams for all of these views.

## Why Model Software?

“We build models so that we can better understand the system we are developing.” (Booch UML User Guide page 6)

Specifically, modeling enables you to:

- Visualize new or existing systems  
Models help us understand, conceptualize, and visualize any kind of system. You can model systems that already exist. For example, the policies and procedures of a business form an existing business process. A Use Case diagram is used to capture the business “system.”  
You can model systems that have not been built yet, such as the Solution model of the software.
- Communicate decisions to the project stakeholders  
Some models can be easily understood by the business owner, domain experts, and other client-side stakeholders. Other models are more relevant to communication among development team members.
- Document the decisions made in each OOSD workflow  
Depending on the size of the development team, not all developers are involved in each workflow. For example, sometimes only the Project Manager and the System Architect will gather requirements. This information (in the form of a Requirements model) must be communicated to the design team.
- Specify the structure (static) and behavior (dynamic) elements of a system  
During the Design and Implementation workflows, the structural and dynamic aspects of the software system must be specified from the Requirements model. The Solution model represents the

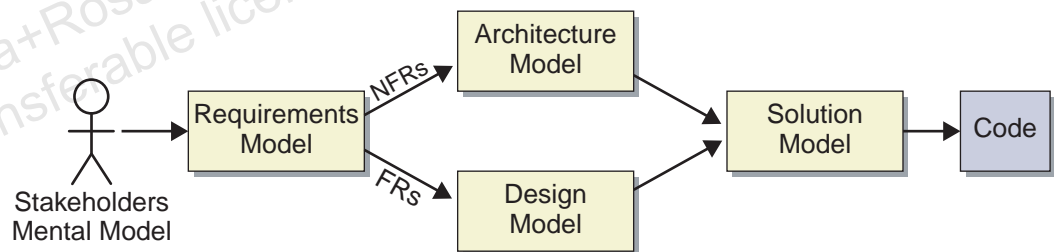
complete conceptualization of the software system. The UML provides various diagrams that support creating these types of views of the Solution model.

- Use a template for constructing the software solution

The Solution model is used as a template for implementing the code modules that make up the software system.

## OOSD as Model Transformations

Figure 2-9 shows a few of the models that a development team might create. The project starts as the mental models of the client-side stakeholders. During the Requirements Gathering and Analysis workflows the mental models are combined and transformed into the *Requirements model*. The non-functional requirements of the Requirements model are transformed in the *Architecture model*, which defines the high-level structure of the software solution. The functional requirements of the Requirements model are transformed into a *Design model*, which defines the abstract components of the software solution. The Design model is merged with the Architecture model to produce the *Solution model*, which defines the detailed structure of the software solution. The Solution model is used to guide the construction of the code for the software solution.



**Figure 2-9** Software Development as a Series of Model Transformations

Some of these models are important to capture and record in an artifact. Some artifacts are documents. Some artifacts are diagrams of visual models of the system. Finally, some artifacts are the program modules that make up the implementation.

In this section, you will be briefly introduced to the Unified Modeling Language which this course uses to create diagram artifacts.



**Note** – Not all methodologies put a high value on creating artifacts of models. eXtreme Programming (XP) is such a methodology. In XP, the requirements are captured by user *stories*, and the *design* is captured by the current state of the code. However, mental models at various stages of abstraction will always exist in the minds of developers. With the techniques taught in this course you should be able to create and represent your mental models in the UML.

---

## Defining the UML

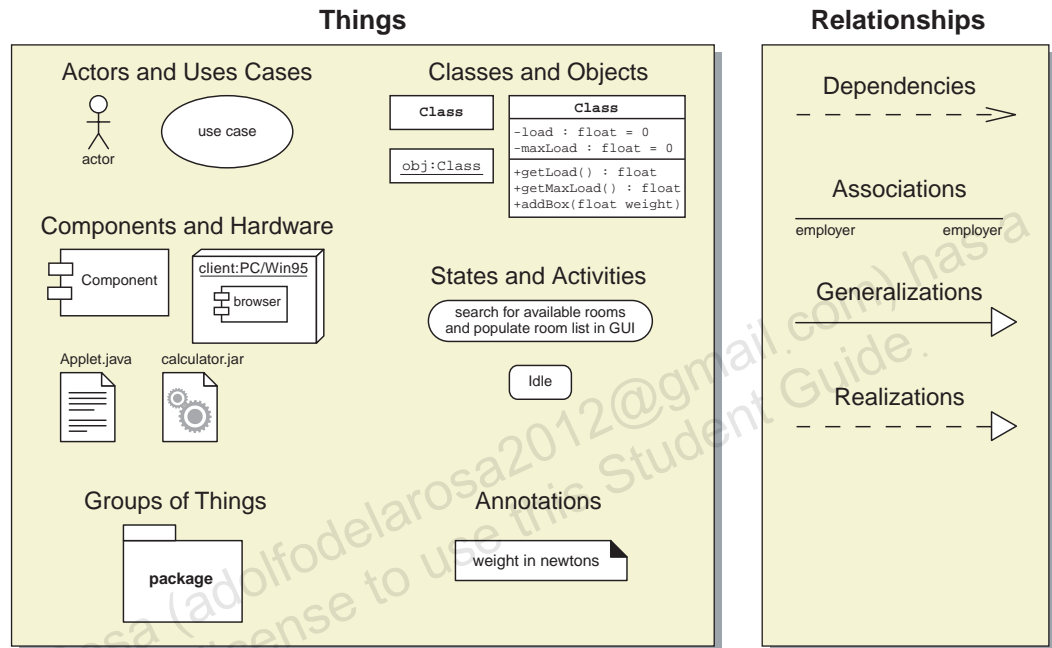
“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.” (UML v1.4 page xix)

Using the UML, a model is composed of:

- Elements (things and relationships)
- Diagrams (built from elements)
- Views (diagrams showing different perspectives of a model)

## UML Elements

UML diagrams are built from modeling primitives or elements. There are two broad categories of elements: things (also called nodes) and relationships (also called links). Figure 2-10 shows many of the fundamental UML elements.

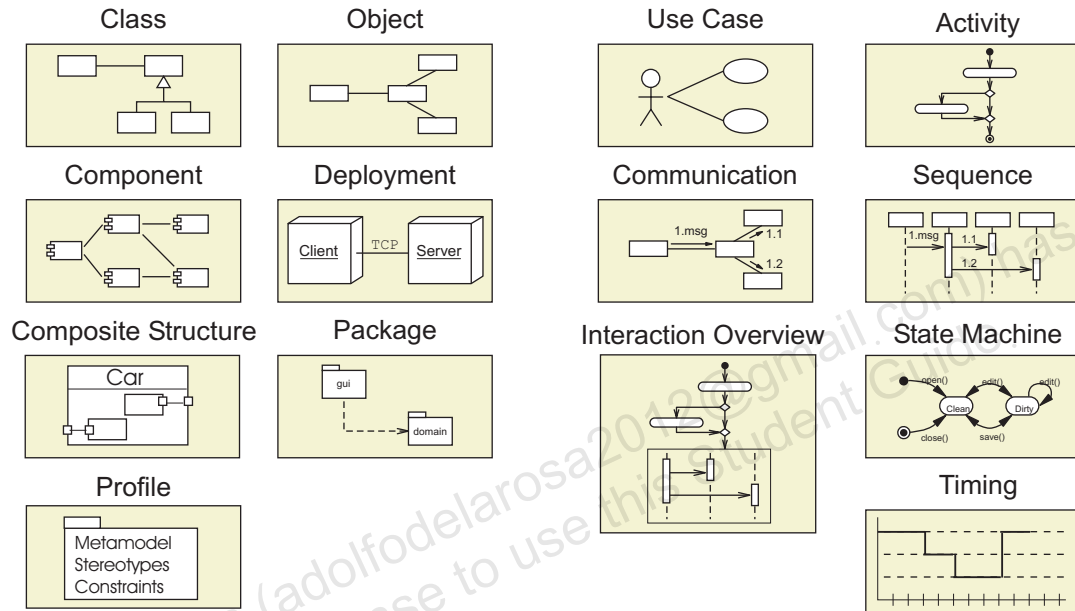


**Figure 2-10** Elements of UML Diagrams

Throughout the course, you will be introduced to each UML diagram and all of the elements that make up that diagram.

## UML Diagrams

UML diagrams enables you to create visualizations of your mental models of a software system. These diagrams are used to construct many of the artifacts in the workflows described in this course. Figure 2-11 shows there are currently 14 fundamental types of diagrams in UML 2.2.



**Figure 2-11** Set of UML Diagrams

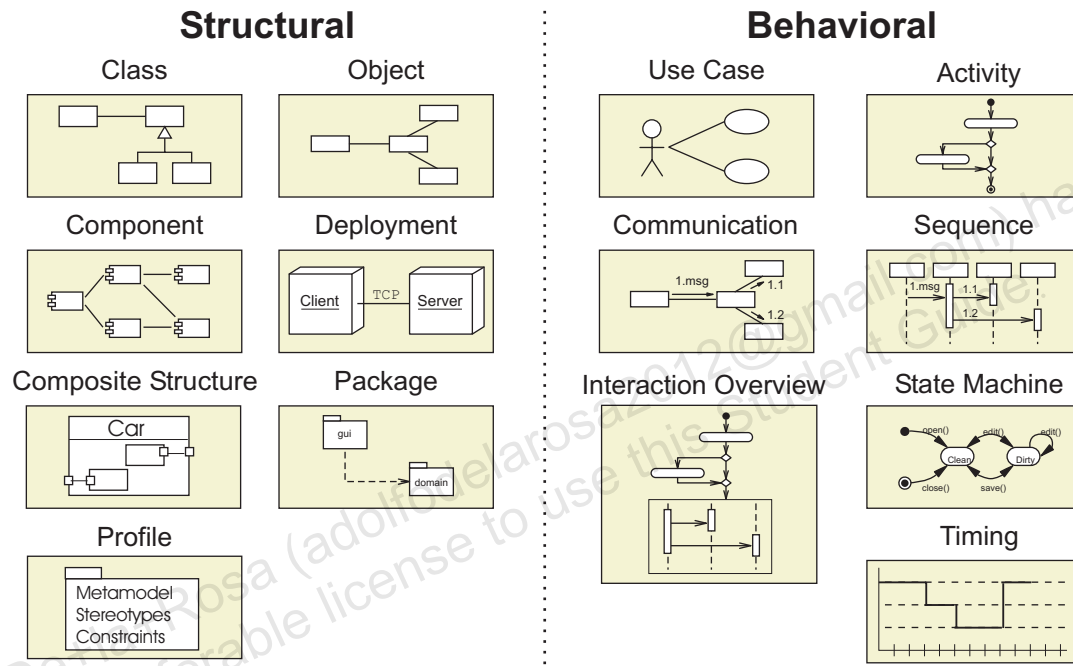
Following is a brief description of each diagram:

- A *Use Case diagram* represents the set of high-level behaviors that the system must perform for a given actor.
- A *Class diagram* represents a collection of software classes and their interrelationships.
- An *Object diagram* represents a runtime snapshot of software objects and their interrelationships.
- A *Communication diagram* (formerly *Collaboration diagram*) represents a collection of objects that work together to support some system behavior.
- A *Sequence diagram* represents a time-oriented perspective of an object communication.
- An *Activity diagram* represents a flow of activities that might be performed by either a system or an actor.

- A *State Machine diagram* represents the set of states that an object might experience and the triggers that transition the object from one state to another.
- A *Component diagram* represents a collection of physical software components and their interrelationships.
- A *Deployment diagram* represents a collection of components and shows how these are distributed across one or more hardware nodes.
- A *Package diagram* represents a collection of other modeling elements and diagrams.
- An *Interaction Overview diagram* represents a form of activity diagram where nodes can represent interaction diagram fragments. These fragments are usually sequence diagram fragments, but can also be communication, timing, or interaction overview diagram fragments.
- A *Timing diagram* represents changes in state (state lifeline view) or value (value lifeline view). It can also show time and duration constraints and interactions between timed events.
- A *Composite Structure diagram* represents the internal structure of a classifier, usually in form of parts, and can include the interaction ports and interfaces (provided or required).
- A *Profile diagram* might define additional diagram types or extend existing diagrams with additional notations.

## UML Diagram Categories

The UML diagrams can be categorized into two main categories: structural (show the static structure of the objects in a system) and behavioral (show the dynamic behavior of objects in a system). Figure 2-12 groups the UML diagrams into these two main categories. It further shows that four of the behavioral diagrams can be subcategorized as interaction diagrams.



**Figure 2-12** Views Created From UML Diagrams

**Note** – A few of these UML 2.2 diagrams will not be formally covered in this course.

## Common UML Elements and Connectors

UML has a few elements and connectors that are common across UML diagrams. These elements and connectors include:

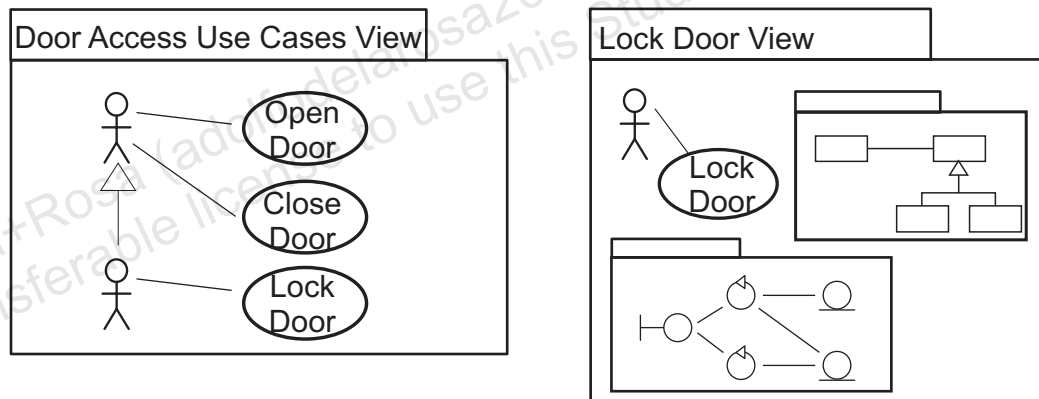
- Package
  - A package is used to group together any UML elements and diagrams.
  - Packages are elements, therefore they can be nested.



- A package is a logical view. Therefore, an element might exist in more than one package to form different views.
- Java technology packages are a subset of UML packages.

A package in UML is used to group together elements (which includes packages) or diagrams to form a cohesive logical view. These packages might be named to provide a namespace for these groups. Packages might be nested to provide a hierarchical logical or conceptual view. Most packages do not relate to any physical grouping. Because they are logical, an element can exist within more than one package to form different views of that element. For example, a use case might be viewed related to other use cases, and it might also be shown packaged with the classes it interacts with.

Figure 2-13 shows an example of using packages for views. The Door Access Use Cases View shows the use cases in the system. The Lock Door View shows the Lock Door use case, along with a subset of classes that are used by that use case and one or more interaction diagrams that show the scenarios of the use case.

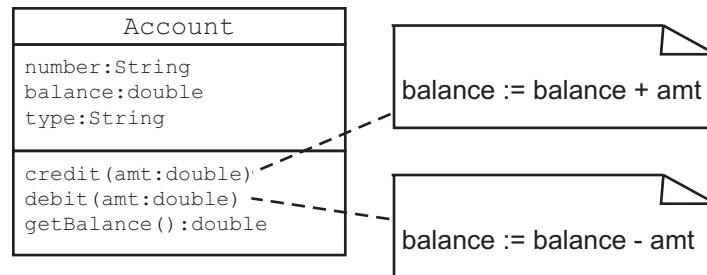


**Figure 2-13** Example of Using Packages

- Note

A note allows textual notes to be added to any aspect of a diagram. This allows additional information in the form of text to be attached (with a dashed line) to any UML element. For example, notes can annotate classes, methods, components, actors, associations, and so on. A note in UML is in the form of a dog-eared rectangle with its upper-right corner bent over. Figure 2-14 on page 2-32 shows using

UML notes to show the internal behavior of the details of the methods. In UML, colon equals (:=) is an assignment; however, you could have written Java technology syntax instead.



**Figure 2-14** Example of Using a UML Note

- Dependency
  - The dependency notation shows that one UML element depends on another UML element.
  - The type of dependency can be attached to the line with a stereotype.

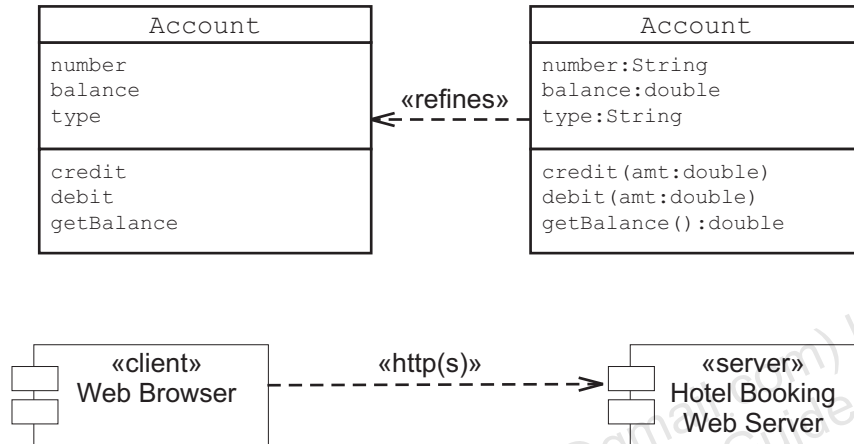


**Caution** – This notation of a dashed line with an open arrowhead has an alternative meaning in Activity diagrams.

- Stereotypes
 

Stereotypes are used to specify a more specific type of element or connector type. The notation for stereotypes is to bracket them in a Guillemont quotation marks symbols « » used in French and some other languages. However, for the typographically challenged, double angle brackets are a suitable substitute.

Figure 2-15 shows an example of an Account class in its analysis form with minimal details, and then in its design (refined) form with design details added. The figure also shows an example of a Web Browser depending on a Web Server and the dependency stereotype showing the protocol as http or https.



**Figure 2-15** Example of UML Dependencies and Stereotypes

Table 2-6 shows some common UML stereotypes that are used in the course. For example an «interface» stereotype can be used on a UML class to specify that it only has OO interface (method signature definitions), but no implementation.

Table 2-7 shows some common stereotypes that were use in this course that are not part of the UML.

UML examples make extensive use of stereotypes. Table 2-6 lists some of standard UML stereotypes used in this course.

**Table 2-6** Standard UML Stereotypes

Stereotype	Definition
«actor»	<i>An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates. (UML v1.4 page 3-97)</i>
«create»	<i>Create is a stereotyped usage dependency denoting that the client classifier creates instances of the supplier classifier. (UML v1.4 page 2-52)</i>

**Table 2-6** Standard UML Stereotypes (Continued)

Stereotype	Definition
«extend»	<i>An extend relationship from use case A to use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behavior specified by A. (UML v1.4 page 3-98)</i>
«import»	<i>Import is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target package are added to the namespace of the source package. (UML v1.4 page 2-48)</i>
«include»	<i>An include relationship from use case E to use case F indicates that an instance of the use case E will also contain the behavior as specified by F. (UML v1.4 page 3-98)</i>
«interface»	<i>An interface is a specifier for the externally-visible operations of a class, component, or other classifier (including subsystems) without specification of internal structure. (UML v1.4 page 3-50)</i>
«refine»	<i>Specifies refinement relationship between model elements at different semantic levels, such as analysis and design. (UML v1.4 page 2-18)</i>

Table 2-7 lists some of the stereotypes that were created for this course.

**Table 2-7** Non-Standard UML Stereotypes

Stereotype	Definition
«accessors»	This stereotype delineates the set of methods that retrieve the attributes of a class.
«operations»	This stereotype delineates the set of operations for an Analysis-level class.
«constructors»	This stereotype delineates the set of Java technology constructors of a class.
«mutators»	This stereotype delineates the set of methods that change or set the attributes of a class.
«UI Frame»	A class that denotes a graphical user interface (GUI) frame or window.

**Table 2-7** Non-Standard UML Stereotypes (Continued)

Stereotype	Definition
«tcp/ip»	This stereotype denotes the use of Transport Control Protocol/Internet Protocol (tcp/ip) to communicate between hardware nodes.
«http(s)»	This stereotype denotes the use of Hypertext Transfer Protocol (http) or Secure Hypertext Transfer Protocol (https) to communicate between components.
«methods»	This stereotype delineates the set of business logic methods of a class.
«Controller»	This stereotype denotes a component that acts as a user interface (UI) controller.
«View»	This stereotype denotes a component that acts as a visual UI component.
«Service»	This stereotype denotes a component that manage interactions between collections of objects. This is basically the same as the «control» stereotype defined in the “UML Profile for Software Development Processes.”
«Entity»	This stereotype denotes a component that acts as a persistent Domain entity. This is basically the same as the «entity» stereotype defined in the “UML Profile for Software Development Processes.”
«JDBC»	This stereotype denotes the use of the Java Database Connectivity™ technology protocol to communicate between a component and a Relational Database Management System (RDBMS).
«Java package»	A package that holds a set of Java technology class components.

## What UML Is and Is Not

There are many misconceptions about UML. Table 2-8 lists some of the more significant misconceptions.

**Table 2-8** Misconceptions About the UML

UML is not:	But it:
Used to create an executable model.  Models are not executable. Only working code is executable.	Can be used to generate code skeletons.  UML tools can generate the software skeletons from a model. There are attempts to produce executable UML in the future.
A programming language.  It is a visual, modeling language.	Maps to most OO languages.  UML has features that enable a software designer to represent programming language constructs.
A development process (methodology).  UML is simply a set of diagrams that can be used during the development process to model and document. It does not define a process that tells you when and how to build these diagrams.	Can be used as a tool within the activities of a development process (methodology).  UML is a fundamental tool to several popular OO development processes, including the Unified Process. UML can even be used if you do not use a formal development process.

### UML Tools

UML itself is a tool. You can create UML diagrams on paper, on a white boards, on flip-chart-sized plastic sheets (which are statically charged, will adhere to most surfaces, and can be written on using whiteboard markers). For example, you can create models by drawing on napkins over lunch; have team meetings to collaborate on a Class diagram on a conference room whiteboard; or paper the walls and windows of a room with numerous plastic sheets, draw on these sheets, and then move these sheets to another room. This enables a great deal of flexibility in the creation of models and team collaboration. But when a model must be captured as a long-lived artifact, then the diagram should be drawn in a tool that will enable printing and archiving.

UML tools can be divided into roughly two categories: drawing and modeling tools. Tools such as StarOffice, Illustrator, and Visio can draw UML diagrams, but using such tools puts the burden on the developer to verify that the diagram is drawn correctly.

UML modeling tools:

- Provide computer-aided drawing of UML diagrams  
For example, UML modeling tools may prohibit you from placing an actor node in an Object diagram. This verification is accomplished by restricting the drawing operations of the UML tool by the syntactic constraints of the UML specification. This restriction may be relaxed because UML 2.2 does not strictly enforce the boundaries between diagrams. Therefore, it should be possible to include a state machine inside an internal structure.
- Support (or enforce) semantic verification of diagrams  
For example, UML modeling tools will maintain a semantic connection between a class called *xyz* in one Class diagram and the same *xyz* class in a Component diagram.  
This referential integrity is accomplished by maintaining a single, consistent model underlying the diagrams drawn in the modeling tool.
- Provide support for a specific methodology  
For example, Rational Software Modeler (RSM) and Rational Software Architect (RSA) include UML tools that supports the Rational Unified Process (RUP). This is a benefit if your team has chosen the methodology used by the tool that the team purchased. Otherwise, the tool might be a liability.

---

**Note** – RSA and RSM supersede the well-know product Rational Rose.

---



- Generate code skeletons from the UML diagrams  
Most UML tools can generate code skeletons for a variety of OO languages, such as C++ and the Java programming language. This process of generating code from models is called *forward engineering*. Some tools can also generate UML diagrams from existing source code. This is called *reverse engineering*. Some tools can keep the diagrams and source code synchronized.
- Organize all of the diagrams for a project

Keeping track of the multitude of diagrams for a large software project can be a major task with ordinary drawing tools. UML modeling tools provide built-in support for maintaining all UML diagrams for a project. Some tools also provide built-in support for version control.

- Automatic generation of modeling elements for design patterns, Java™ Platform, Enterprise Edition (Java™ EE platform) components, and so on

Some UML tools provide advanced mechanisms that fill in templates of design patterns and other frameworks, such as Java EE platform components.



## Summary

In this module, you were introduced to OOSD at a very high level. Here are a few important concepts:

- The OOSD process starts with gathering the system requirements and ends with deploying a working system.  
A working, tested, and deployed (production) system is the ultimate goal of the OOSD process. Everything that you do should be to support the creation of a working system.
- Workflows (disciplines) define the activities that transform the artifacts of the project from the use cases to the implementation code (the final artifact).  
The workflows and their activities lead you from the Requirements model to the production system.
- The UML supports the creation of visual artifacts that represent views of your models.  
Use UML as your primary tool to create visual representations of the models you build throughout the OOSD process.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Creating Use Case Diagrams

---

## Objectives

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

- Justify the need for a Use Case diagram
- Identify and describe the essential elements in a UML Use Case diagram
- Develop a Use Case diagram for a software system based on the goals of the business owner
- Develop elaborated Use Case diagrams based on the goals of all the stakeholders
- Recognize and document use case dependencies using UML notation for extends, includes, and generalization
- Describe how to manage the complexity of Use Case diagrams by creating UML packaged views

## Additional Resources

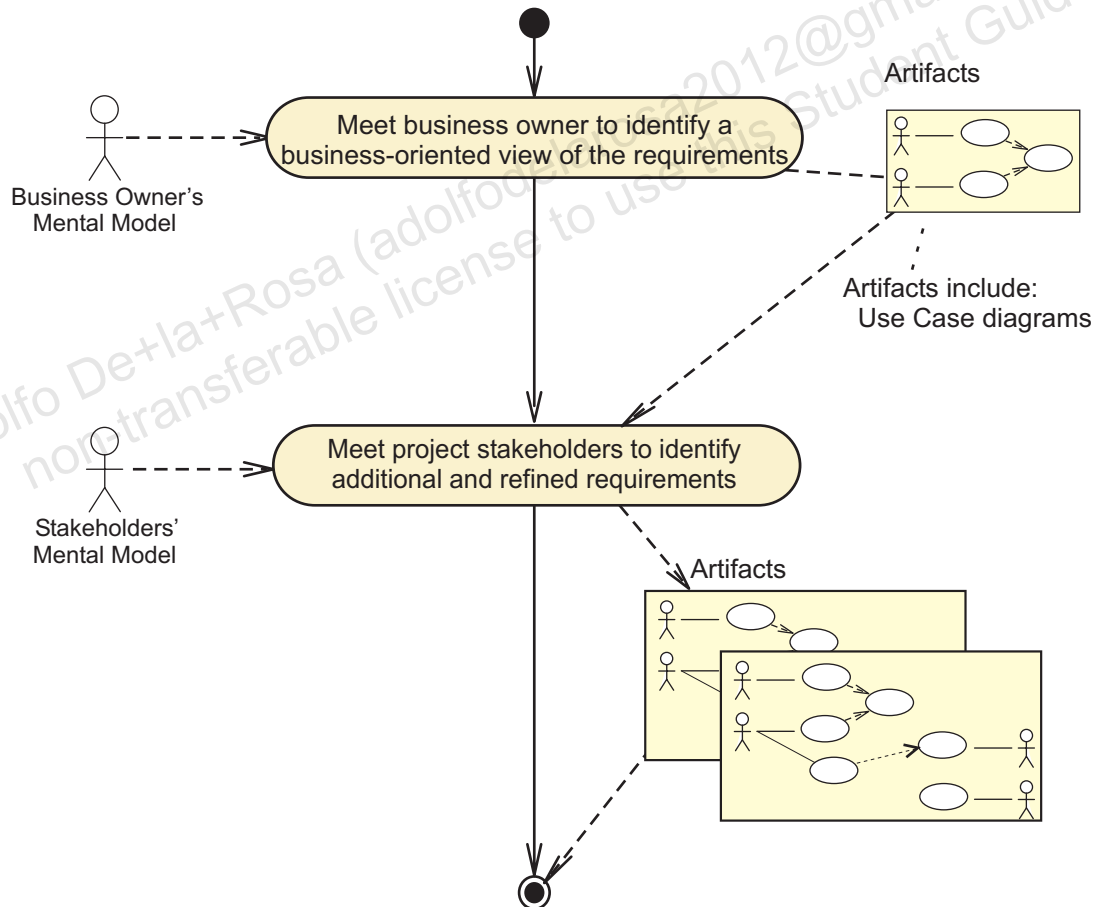
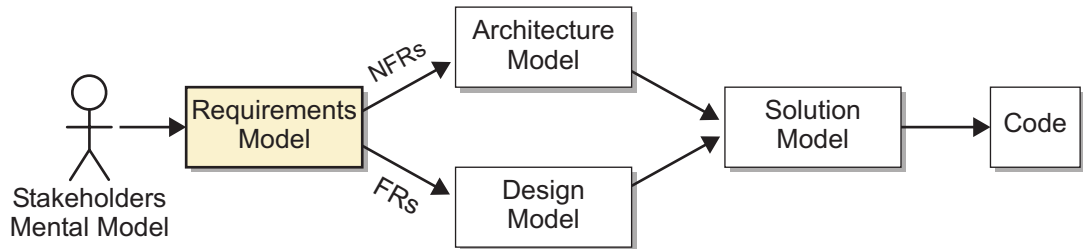


**Additional resources** – The following references provide additional information on the topics described in this module:

- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Folwer, Martin, Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- The Object Management Group. “Unified Modeling Language (UML), Version 2.2”  
[<http://www.omg.org/technology/documents/formal/uml.html>].
- Rosenberg, Doug, Kendall Scott. *Use Case Driven Object Modeling with UML (A Practical Approach)*. Reading: Addison Wesley Longman, Inc., 1999.
- Larman, Craig. *Applying UML and Patterns (3rd ed)*. Upper Saddle River: Prentice Hall, 2005.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004.

# Process Map

This module covers the next step in the Requirements Gathering workflow: Creating the initial Use Case diagram. Figure 3-1 shows the activity and artifact covered in this module.



**Figure 3-1** Requirements Gathering Process Map

## Justifying the Need for a Use Case Diagram

This module describes how to use a Use Case diagram to model and document the goals (use cases) of its users. The following points explain the purpose of a Use Case diagram:

- A Use Case diagram enables you to identify—by modeling—the high-level functional requirements (FRs) that are required to satisfy each user's goals.

A Use Case diagram provides a visual representation of the high-level FRs. The Use Case diagram is often easier to model with the client-side stakeholders than the alternative textual representation.

- The client-side stakeholders need a big-picture view of the system.

A Use Case diagram provides a high-level view of the entire system. All non-trivial systems will have too many use cases to view at the same time. The use of UML packages allows you to see a high-level view of the packages, each containing either use cases or subpackages. The packages are simply views of related use cases that can be categorized in different ways—for example, Sales, Marketing, and Shipping. This view can be the basis of a common language between the client-side stakeholders and the development team.

- The use cases form the basis from which the detailed FRs are developed.

Use cases are the central focus of system development. A Use Case diagram is the high-level guide for the development team. The lower-level functionality of the system can be identified by exploring the internal behavior of each use case, which will be covered in the following module.

- Use cases can be prioritized and developed in order of priority.

Use cases can be assigned priorities based on business need, complexity, and dependency on other use cases. In an incremental development process, the priority affects the iteration in which the use case will be developed.

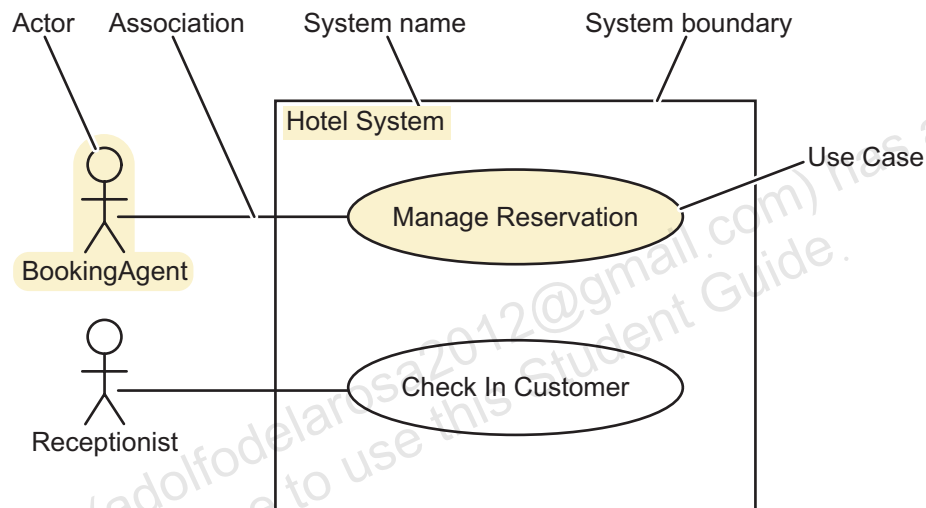
- Use cases often have minimal dependencies, which enables a degree of independent development.

Because use cases often have minimal dependencies, they can be developed in parallel or by more specialist teams.

## Identifying the Elements of a Use Case Diagram

A Use Case diagram shows the relationships between actors and the goals they wish to achieve.

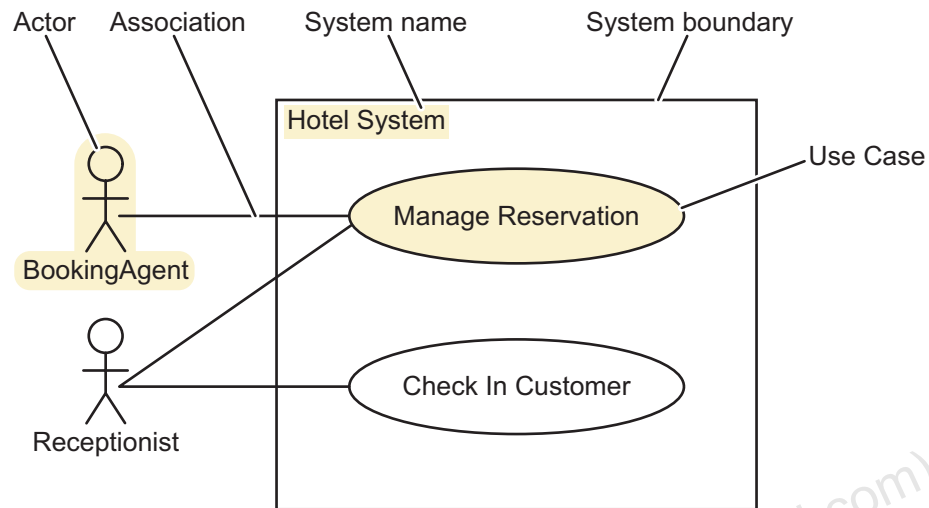
A *Use Case diagram* provides a visual representation of the system, the use cases that the system provides, and the actors (job roles) that use the system to perform specific functions. Figure 3-2 shows an example Use Case diagram.



**Figure 3-2** An Example Use Case Diagram

An alternative style is to draw an association line between the Receptionist actor and the Manage Reservation use case as the Receptionist can manage reservations as well.

Actors are simply roles. Any physical person, system, or device can assume multiple roles. Therefore, the job title Receptionist can assume the Receptionist role and also the Booking Agent role. If this was not the case, then the Duty Manager actor would need associations to most of the use cases in the hotel, as would the Hotel Manager and other actors.



**Figure 3-3** An Example Alternate Style Use Case Diagram



## Actors


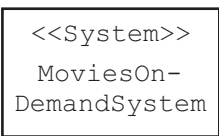

“An actor is a role that a user plays with respect to the system.”  
(Fowler UML Distilled page 42)

“An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., “role”) of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances. (UML Superstructure Specification, v2.2)

**Note** – The subject is the system under consideration to which the use cases apply.

Anyone or anything that is external to the system and interacts with the system is an *actor*. There are fundamentally three classes of actors: people, external systems or devices, and time. Figure 3-4 illustrates these three actor types.

**Table 3-1**

 BookingAgent	 <<System>> MoviesOn-DemandSystem	 Time
<p>This icon represents a human actor (user) of the system.</p>	<p>This icon can represent any actor, but is usually used to represent external systems, devices, or time.</p>	<p>This icon represents a time-trigger mechanism that activates a use case.</p>

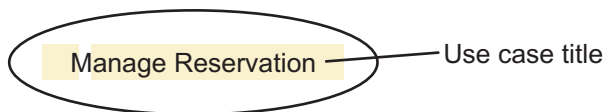
**Figure 3-4** Actor Types

- Actors that initiate and control the use case are subcategorized as primary actors.  
These actors will participate for the entire duration of the use case.
- Actors that are used by a use case are subcategorized as secondary actors.  
These actors will generally participate for only part of the duration of the use case.

## Use Cases

A use case is “The specification of a sequence of actions, including variants, that a system can perform.”

A *use case* describes an interaction between an actor and the system to achieve a goal. Figure 3-5 shows an example use case.



**Figure 3-5** An Example Use Case

- A use case encapsulates a major piece of system behavior with a definable outcome.

A use case provides a visual encapsulation of all of the detailed actions involved in a major system behavior. The use case title provides a communication tool for stakeholders to discuss the system at a high-level.

- A use case is represented as an oval with the use case title in the center.

Alternatively, the title can be placed under the oval.




---

**Note** – Use case title and use case name are synonymous.

---

- A good use case title should consist of a brief but unambiguous verb-noun pair.
  - “Check In” has a verb, but no noun.
  - “Check In Customer when they arrive at the hotel” is too detailed.
  - In tennis, “Enter Score” is a verb-noun pair. However, “Enter Final Match Score” or “Enter Current Set Score” would reduce the ambiguity.

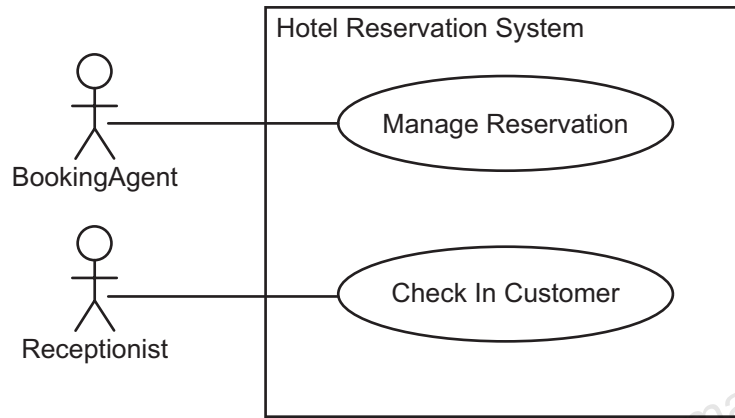
If a verb-noun pair is ambiguous, you will need to use a verb-noun pair phrase to reduce the ambiguity. However, it should be as brief as possible. Also, the terms used should be included in the glossary of terms.

- Use case internal descriptions can be written in a User Interface (UI) independent form.

This allows use cases to be more re-usable and accommodate new UIs. It also allows you to focus on the use case's business process instead of how the UI works. However, this approach might not always be suitable.

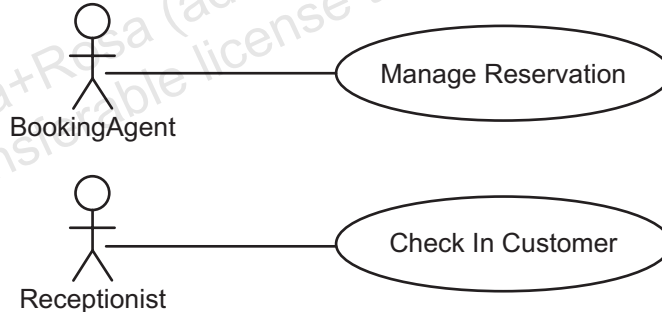
## System Boundary

A Use Case diagram is usually shown with a system boundary. Figure 3-6 illustrates the system boundary.



**Figure 3-6** A Use Case Diagram With a System Boundary

A Use Case diagram may also be drawn without a system boundary. Figure 3-7 illustrates an example of a Use Case diagram without a boundary.



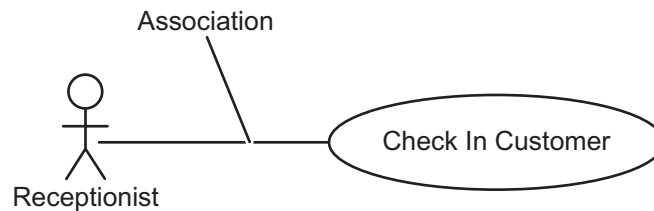
**Figure 3-7** A Use Case Diagram Without a System Boundary

For clarity, you should use a system boundary when drawing a Use Case diagram.

## Use Case Associations

A use case association represents “the participation of an actor in a use case.” (UML v1.4 spec. page 357)

In the Hotel Reservation System, a receptionist role uses the System to perform the Check In Customer use case. Figure 3-8 shows this association.



**Figure 3-8** An Example Use Case Association

- An actor must be associated with one or more use cases.  
An actor with no use case associations does not interact with the system, so there is no reason to represent the actor in the diagram.
- A use case must be associated with one or more actors.  
However, later sections in this module describe that other use cases might be identified that are included in the activity of an essential use case; these sub-use cases might not be directly associated with an actor because they implicitly interact with the included (sub) use case.
- An association is represented by a solid line usually with no arrowheads.  
Some UML tools put an arrow on one end of the association line by default. Some people use this arrow notation to distinguish between the interactions initiated by a primary actor and those initiated by a use case on a secondary actor.

**Note** – Models evolve. It is likely that your first attempt in creating a Use Case diagram of a system will not be final. As the system evolves (through iterative development), you will need to update and refine the Use Case diagram.



## Creating the Initial Use Case Diagram

One of the primary aims of the initial meeting with the project's business owner is to identify the business-significant use cases.

- A use case diagram may be created during the meeting  
This may achieve faster and more accurate results by using the benefits of UML modeling to assist in identifying the major business-oriented goals that the system will achieve.
- Alternatively, the diagrams can be created after the meeting from textual notes.  
The goals can be identified and listed, usually in textual form, during the meeting.

The following text provides an abstract from the meeting with the business owner:

The booking agent (internal staff) must be able to manage reservations on behalf of customers who telephone or e-mail with reservation requests. The majority of these requests will make a new reservation, but occasionally they will need to amend or cancel a reservation. A reservation holds one or more rooms of a room type for a single time period, and must be guaranteed by either an electronic card payment or the receipt of a purchase order for corporate customers and travel agents. These payment guarantees must be saved for future reference.

A reservation can also be made electronically from the Travel Agent system and also by customers directly via the internet.

The receptionist must be able to check in customers arriving at the hotel. This action will allocate one or more rooms of the requested type. In most cases, a further electronic card payment guarantee is required.

Most receptionists will be trained to perform the booking agent tasks for customers who arrive without a booking or need to change a booking.

The marketing staff will need to manage promotions (special offers) based on a review of past and future reservation statistics. The marketing staff will elaborate on the detailed requirements in a subsequent meeting.

The management need a daily status report, which needs to be produced when the hotel is quiet. This activity is usually done at 3 a.m.

Figure 3-9 shows an initial Use Case diagram for these high-level requirements.

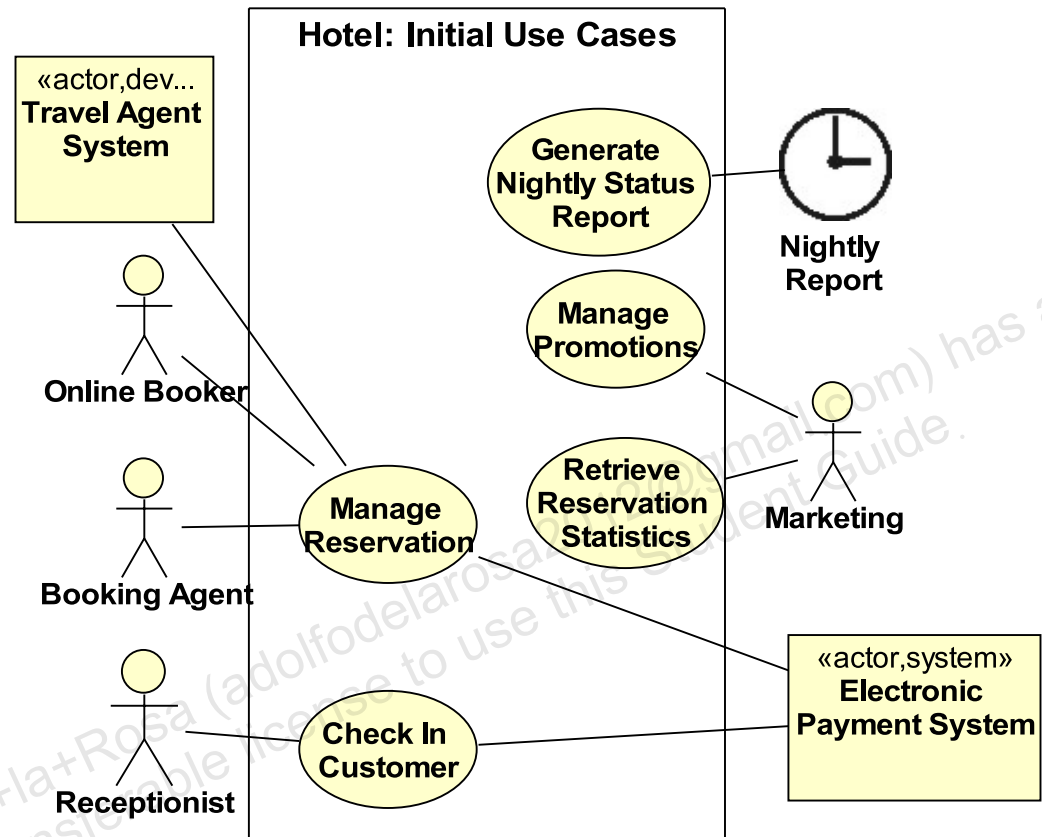


Figure 3-9 Partial Initial Use Case Diagram for a Hotel



## Identifying Additional Use Cases

During the meeting with the business owner, you will typically discover 10 to 20 percent of the use case titles needed for the system.

During the meeting with the other stakeholders, you will discover many more use case titles that you can add to the diagram. For example:

- Maintain Rooms
  - Create, Update, and Delete
- Maintain RoomTypes
  - Create, Update, and Delete

The time of discovery of use cases depends upon the development process.

- In a non-iterative process:
  - You ideally need to discover all of the remaining use case titles, bringing the total to 100 percent.
  - However, this is a resource-intensive task and is rarely completely accurate.
- In an iterative/incremental development process, an option is to:
  - Discover a total of 80 percent of the use case titles in the next few iterations for 20 percent of the effort. This is just one of the many uses of the 80/20 rule.
  - Discover the remaining 20 percent of use case titles in the later iterations for minimal effort.

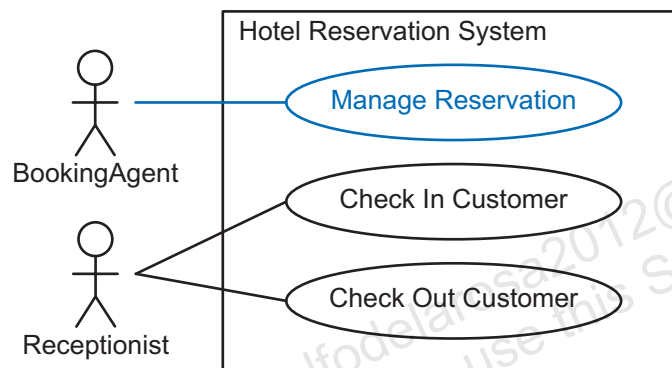
This process works well with software that is built to accommodate change.

The 80/20 rule originates from Pareto Principle, where Vilfredo Pareto created a formula describing the unequal distribution of wealth where 20 percent of the population owned 80 percent of the wealth. This rule applies to many other areas. For example, you can discover 80 percent of the requirement for 20 percent of the effort.

## Use Case Elaboration

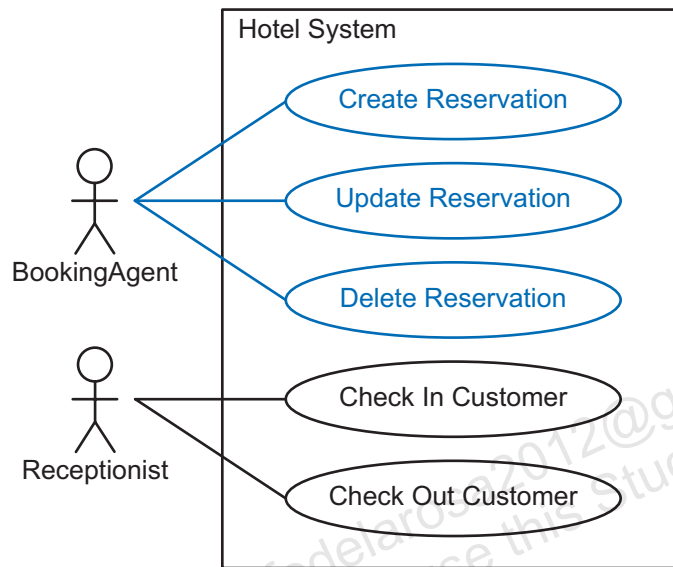
During the meeting with the other stakeholders, you will discover many more use cases that you can add to the diagram. You might also find that some use cases are too high-level. That is, a use case might describe a business function that includes several related workflows.

For example, the Manage Reservation use case is too high-level because it represents several different workflows that all deal with hotel reservations. Figure 3-10 shows this.



**Figure 3-10** An Example High-Level Use Case

In these situations, it is useful to introduce new use cases that separate the workflows. For example, the Manage Reservation use case can be separated into three workflows: one for creating a new reservation, one for updating an existing reservation, and one for deleting an existing reservation. Figure 3-11 shows this.



**Figure 3-11** The High-Level Use Case Separated Into Individual Workflows

Typically, *managing an entity* implies being able to Create, (Retrieve), Update, and Delete an entity (so called, CRUD operations). Other keywords include:

- Maintain
- Process

Other high-level use cases can occur. Identify these by analyzing the use case scenarios and look for significantly divergent flows. Also, if several scenarios have a different starting point, these scenarios might represent different use cases.

Be careful not to refine the use cases too much because use case refinement can become an exercise in functional decomposition and can result in *analysis paralysis*.

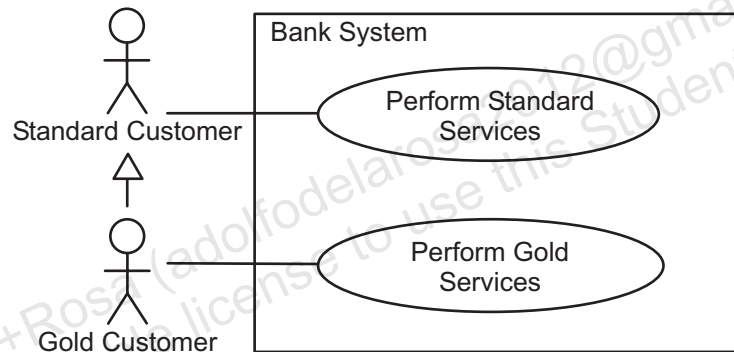
## Analyzing Inheritance Patterns

Inheritance can occur in Use Case diagrams for both actors and use cases:

- An actor can inherit all of the use case associations from the parent actor.
- A use case can be *subclassed* into multiple, specialized use cases.

### Actor Inheritance

An actor can inherit all of the use case associations from the parent actor. For example, the Gold Customer *is a kind of* Standard Customer but can access additional use cases. Therefore, you can use actor inheritance to represent this relationship. Figure 3-12 illustrates this.

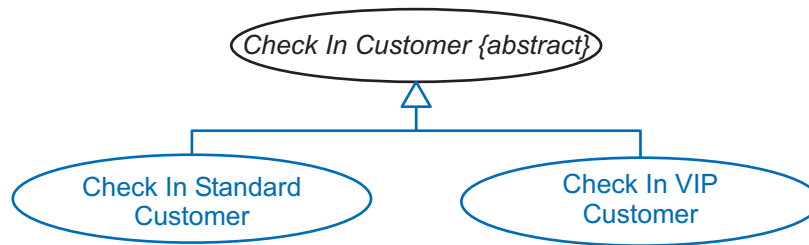


**Figure 3-12** Example Actor Inheritance

This relationship is often used when the phrase “*is a kind of*” does not accurately describe the relationship between the actors. This can often present problems later on, just as it would with class inheritance. For example, if you later (in a later iteration or later version of the software) discover a use case where the Standard Customer can perform a use case that the Gold Customer cannot, you will have to go back and remove this inheritance. This action might require you to make major changes to the software system.

## Use Case Specialization

A use case can be *subclassed* into multiple, specialized use cases. For example, the use case of Retrieve a Reservation can be performed in two distinct ways: search by ID or search by customer. Figure 3-13 illustrates this.



**Figure 3-13** Example Use Case Specialization

Use case specializations are *usually* identified by significant variations in the use case scenarios. These variations are often due to different functional strategies for accomplishing the use case.

The base use case may be marked as abstract, in which case you cannot instantiate the base use case. Therefore in the example in Figure 3-13, you can only instantiate Check In Standard Customer or Check In VIP Customer. If Check In Customer was not abstract, you could instantiate that use case as well.

Using this notation can make the writing of Use Case forms more complicated, therefore is often avoided.

## Analyzing Use Case Dependencies

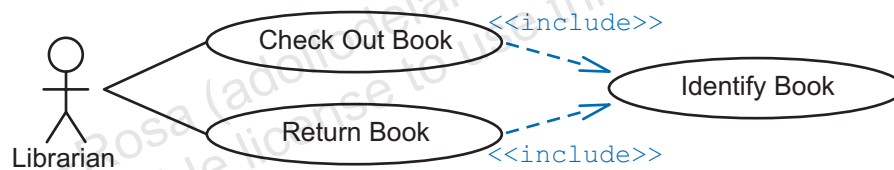
Use cases can depend on other use cases in two ways:

- One use case (a) *includes* another use case (i).  
This means that the one use case (a) requires the behavior of the other use case (i) and always performs the included use case.
- One use case (e) can *extend* another use case (b).  
This means that the one use case (e) can (optionally) extend the behavior of the other use case (b).

### The «include» Dependency

The include dependency enables you to identify behaviors of the system that are common to multiple use cases.

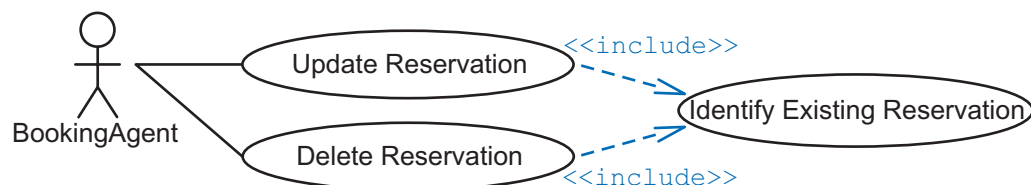
This dependency is drawn with a dependency arrow, and includes the «include» stereotype label. This is illustrated in Figure 3-14.



**Figure 3-14** Example «include» Dependency

To identify common behavior, review the scenarios of multiple use cases for common behaviors. Give this behavior a name and place it in the Use Case diagram with an «include» dependency.

For example, both the Update Reservation and Delete Reservation use cases require the ability to identify the reservation that is being updated or deleted. This would be done if there is significant actor interaction. This behavior can be given a name “Identify Existing Reservation” and placed in the diagram with the appropriate «include» dependencies. Figure 3-15 shows this.



**Figure 3-15** An «include» Dependency in the Hotel Reservation System

Another situation for refining the Use Case model involves interactions with external systems. It is often useful to explicitly represent these interactions with separate use cases.

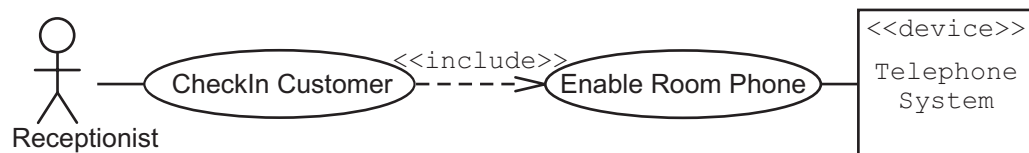
To identify behavior associated with an external actor (system or device), review the scenarios for sequences of behavior that involve an external actor (system or device). Give this behavior a name, and place it in the Use Case diagram with an «include» dependency. Look for other use cases that might include these use cases and record additional dependencies.

For example, the Hotel Reservation System will be connected to an external Hotel Phone System. This system is used when a customer checks in to enable the phone in the room that the customer will occupy. Figure 3-16 shows the Use Case diagram without this «include» dependency. As a result, the reason for the participation of the Telephone System is less obvious.



**Figure 3-16** Example Where the Secondary Actor's Role is not Clear

Figure 3-17 shows the secondary actor's reason for participation in this use case more clearly.

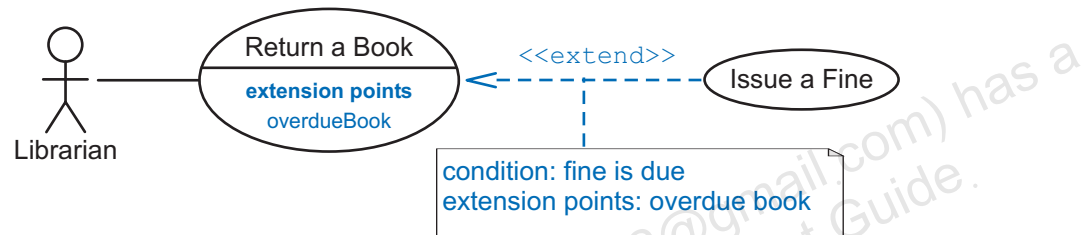


**Figure 3-17** Example Using «include» to Highlight the Secondary Actor's Participation

## The «extend» Dependency

The extend dependency enables you to identify behaviors of the system that are not part of the primary flow, but exist in alternate scenarios.

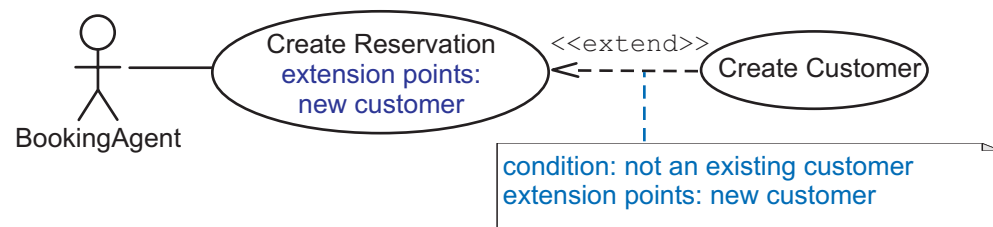
This dependency is drawn with a dependency arrow, an «extend» stereotype label, and an additional label that identifies the “extension point.” The extension point defines the condition within the main use case under which the extension use case is required. Furthermore, the extension points can be listed in the use case node itself. Figure 3-18 illustrates this.



**Figure 3-18** Example Extends Dependency

To identify behaviors associated with an alternate flow of a use case, review the scenarios for sequences of behavior that are optional or could be added in a later increment of the system. Give this behavior a name and place it in the Use Case diagram with an «extend» dependency. Look for other use cases that might be extended by these use cases.

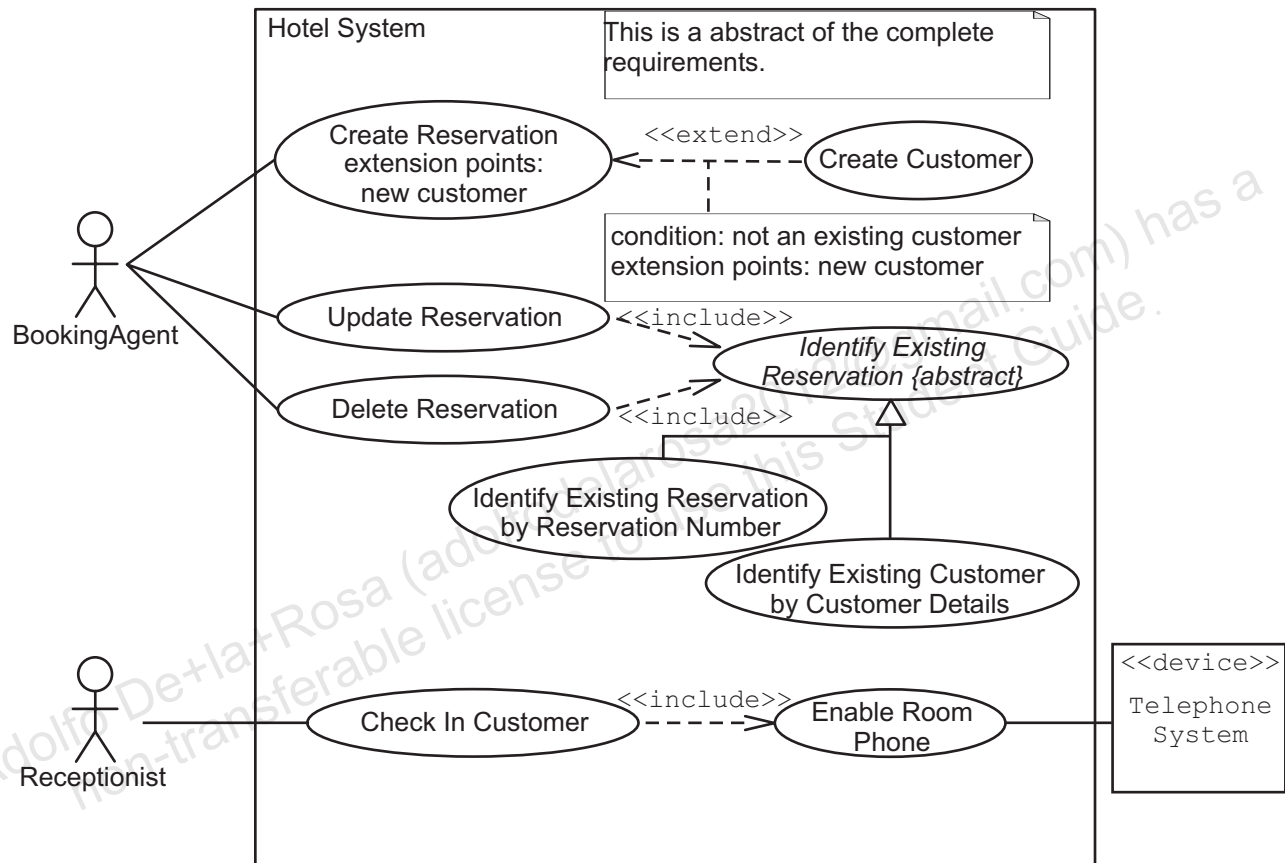
For example, when a reservation is created, the reservation must be associated with a customer. This customer can be an existing customer, which is simply a step in the main or alternate flow. However, if the customer is not an existing customer, you need to perform additional behavior to add the customer. Figure 3-19 shows this.



**Figure 3-19** An «extend» Dependency in the Hotel Reservation System



Figure 3-20 illustrates an example showing a possible refinement of the Use Case diagram for the Hotel Reservation System. In this example, the Identify Existing Reservation by Reservation Number and Identify Existing Customer by Customer Details use cases will have different interactions with the actor. However, if these differences are minor, there might be no need to provide these two specialization use cases.

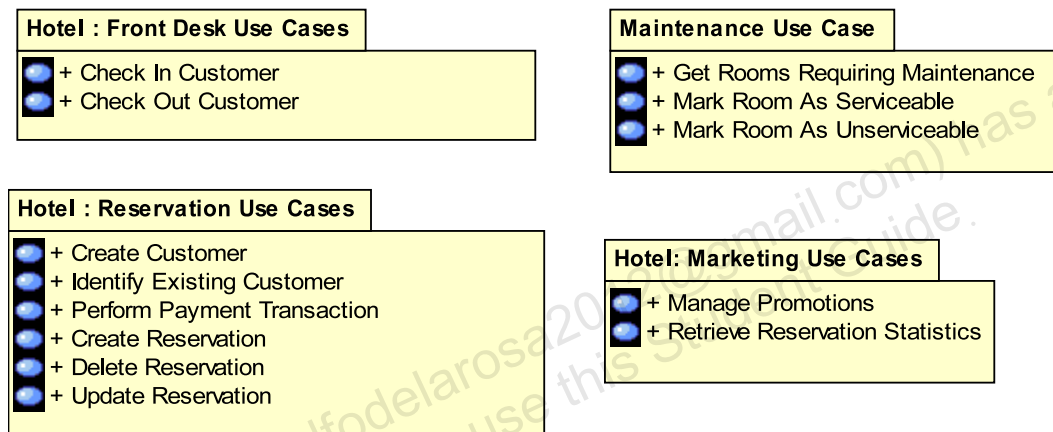


**Figure 3-20** A Combined Example From the Hotel Reservation System

## Packaging the Use Case Views

It should be apparent that any non-trivial software development would need more use cases than could be viewed at one time. Therefore, you need to be able to manage this complexity.

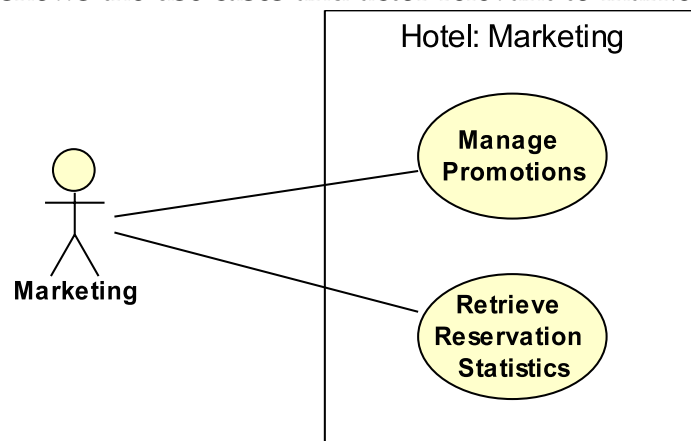
One way of managing this complexity is to break down the use cases into package views. Figure 3-21 shows some of the packages you could create in order to view subsets of the use cases.



**Figure 3-21** Example Showing Packages of Use Case

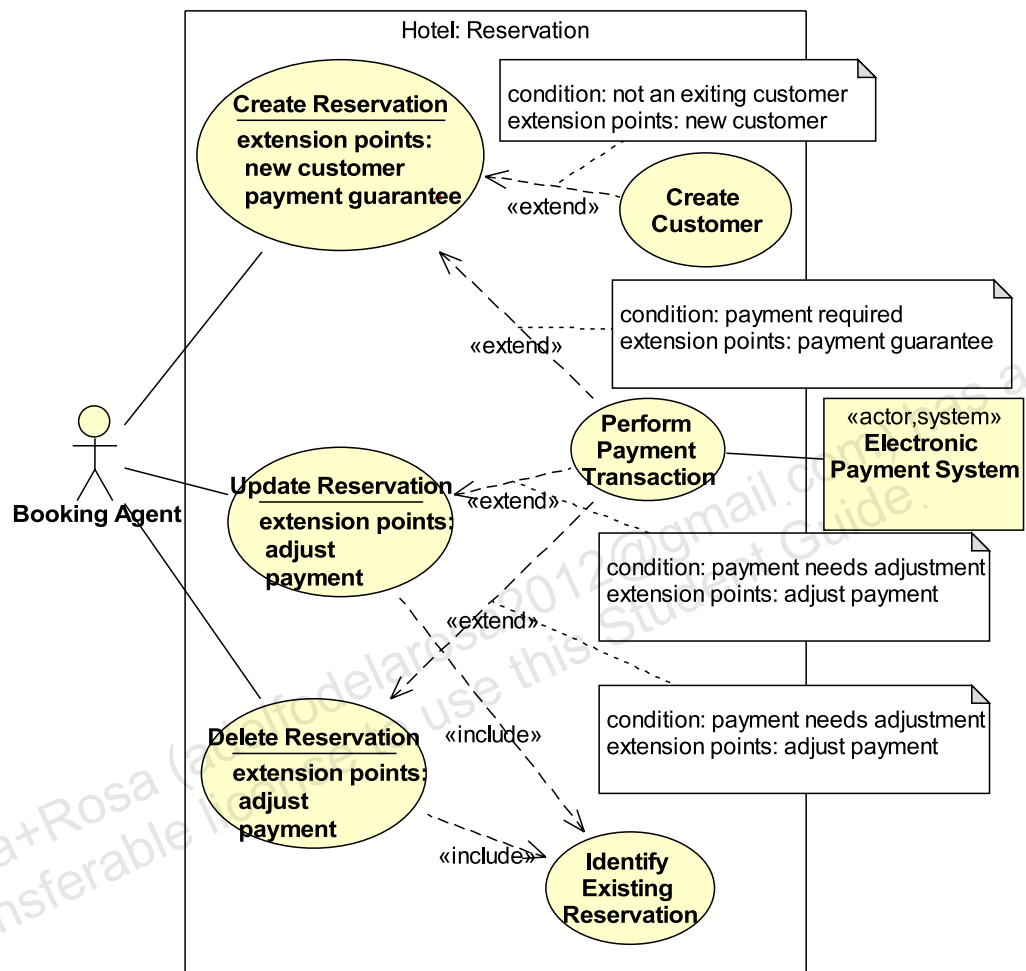
You can look inside each package to reveal the detailed content. Also, a use case element may exist in multiple packages, where it participates in multiple views.

Figure 3-22 shows the use cases and actor relevant to marketing.



**Figure 3-22** Example of Marketing Use Cases and Actor

Figure 3-23 shows some of the use cases and actors relevant to reservations.



**Figure 3-23** Example of Reservation Use Cases and Actors

## Summary

In this module, you were introduced to the UML Use Case diagram and Use Case scenarios. Here are a few important concepts:

- A Use Case diagram provides a visual representation of the big-picture view of the system.
- The Use Case diagram represents the actors that use a system, the use cases that provide a behavior with a definable goal for an actor, and the associations between actors and use cases.
- Use Case diagrams can be elaborated to show a software system based on the goals of the business owner and all the other Stakeholders.
- Use Case diagrams can be elaborated to show use case dependencies by using UML notation for extends, includes, and generalization.
- Complex Use Case diagrams can be broken down into views by using UML packages.

# Creating Use Case Scenarios and Forms

---

## Objectives

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

- Identify and document scenarios for a use case
- Create a Use Case form describing a summary of the scenarios in the main and alternate flows
- Describe how to reference included and extending use cases.
- Identify and document non-functional requirements (NFRs), business rules, risks, and priorities for a use case
- Identify the purpose of a Supplementary Specification Document

## Additional Resources

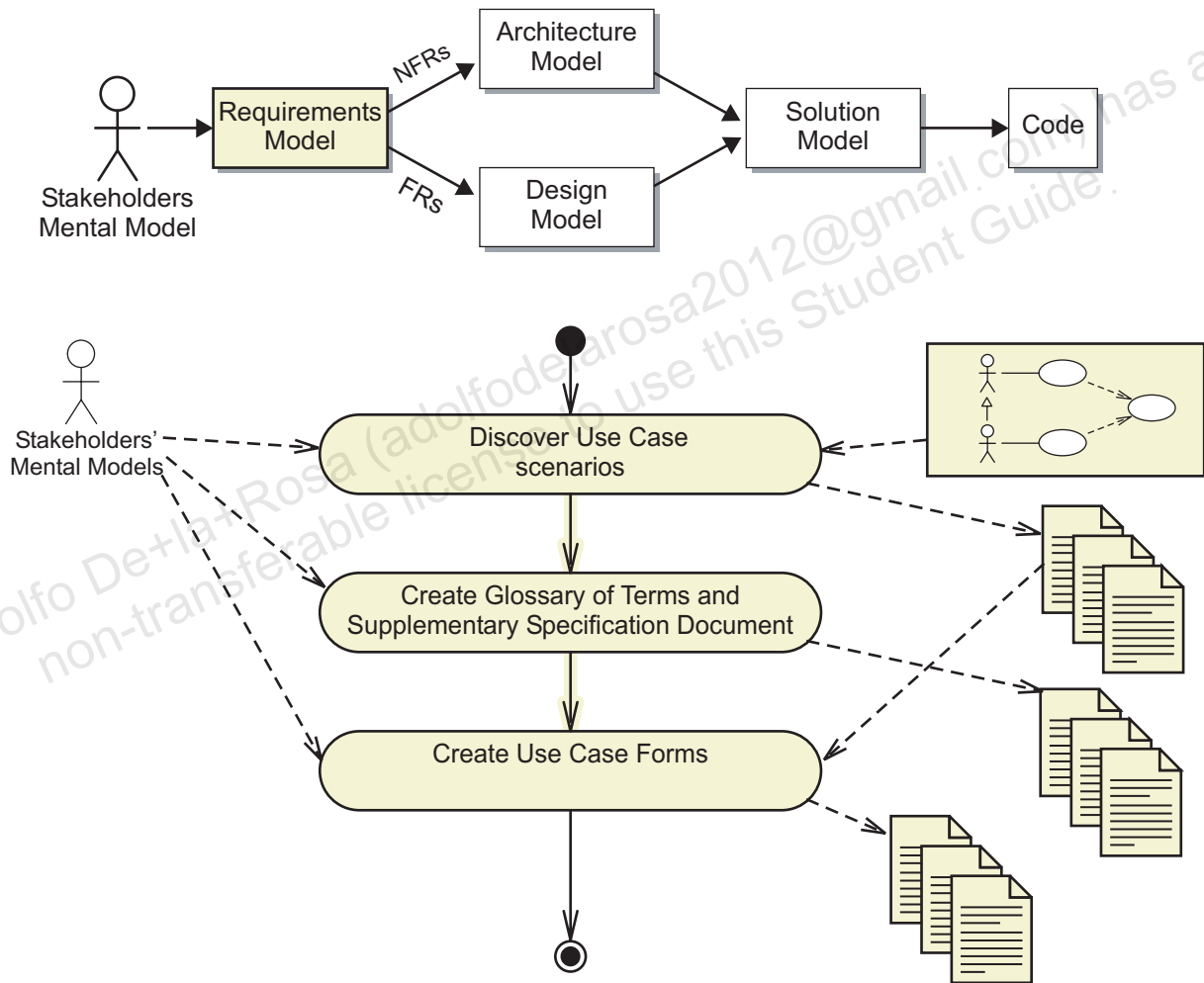


**Additional resources** – The following references provide additional information on the topics described in this module:

- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Harlow: Addison Wesley Longman, Inc., 1993.
- Rosenberg, Doug, Kendall Scott. *Use Case Driven Object Modeling with UML (A Practical Approach)*. Reading: Addison Wesley Longman, Inc., 1999.
- Rosenberg, Doug, Kendall Scott. *Applying Use Case Driven Object Modeling with UML (An Annotated e-Commerce Example)*. Reading: Addison Wesley Longman, Inc., 2001.
- Folwer, Martin, Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- Larman, Craig. *Applying UML and Patterns (3rd ed)*. Upper Saddle River: Prentice Hall, 2005.

# Process Map

This module covers the first steps in the Requirements Gathering and Requirements Analysis workflows. Figure 4-1 shows the activities and artifacts covered in this module.



**Figure 4-1** Use Case Scenarios and Forms Process Map

## Recording Use Case Scenarios

A use case represents a system behavior from the actor's perspective. A *Use Case scenario* is a concrete example of a use case; that is, a single instance of an actor interacting with the use case. Use Case scenarios can be derived from existing business practices or from observing the use of an existing system (which the proposed system is replacing).

A Use Case scenario should:

- Be as specific as possible  
Provide specific information. This will provide a clear and tangible expression of the use case. You may use people's names in the narrative, but only if it adds clarity. These documents should be confidential to the client so it is acceptable and appropriate to use the real names of the client's customers and employees.
- Never contain conditional statements  
You should never have a sentence in the scenario that includes an if statement. For example, this would be unacceptable: "If the customer requests a double room, ask them if they will have additional guests." Instead, create multiple scenarios to cover every case.
- Begin the same way but have different outcomes  
Each scenario for the same use case should begin with the actor in the same state. For example, a booking agent does not begin to create a new reservation in the middle of updating another reservation. To create a reservation, the agent must be waiting for a phone call from a customer.
- Not specify too many user interface details  
The Use Case scenario should focus on the details of the workflow of the use case. It might contain some UI information, but it should not explain all elements of the UI.
- Show successful as well as unsuccessful outcomes (in different scenarios)  
It is important to write scenarios in which the outcomes are unsuccessful as well as those that were successful. Record these different scenarios.

Use Case scenarios drive several other Object-Oriented Analysis and Design (OOAD) workflows. Use Case scenarios are primarily used to create Use Case forms and Test Plans. They may also be used in the creation of Activity diagrams and Object diagrams.



## Selecting Use Case Scenarios

While it is ideal to have multiple scenarios for all use cases, doing so would take a lot of time. Therefore, you can select appropriate use cases for scenario creation by using the following criteria:

- The use case involves a complex interaction with the actor.  
For trivial use cases it might be sufficient to document only a single, successful scenario. For complex use cases, it is important to capture several scenarios, each with a slightly different path through the use case workflow.
- The use case has several potential failure points, such as interaction with external systems or a database.  
For example, one failure point is that the customer's credit card does not have the necessary funds to confirm the reservation. In this scenario, the booking agent can put the reservation on hold until the customer can resolve the credit card problem with the bank or the customer can provide a different credit card. This would be two different scenarios.




---

**Note** – Some methodologies, such as UP, make extensive use of Use Case scenarios. Other methodologies, such as XP, use *user stories*. XP defines the FRs of the system by the complete set of user stories. User stories drive the daily development cycle.

---

There are two types of scenarios:

- Primary (Happy) scenarios record successful results.  
Simple use cases might only have one primary scenario, but complex use cases might have many. Each primary scenario records an event in which the results of the use case were successful. For example, the Manage Reservation use case might include a scenario in which a customer books only one room and another scenario in which a customer books several rooms and uses a promotion for a discount. Success is determined by the client-side stakeholders.
- Secondary (Sad) scenarios record failure events.  
Each secondary scenario records an event in which the results of the use case were unsuccessful. For example, the Manage Reservation use case might include a scenario in which the customer cannot book a room because of the lack of availability or in which the customer's credit card is declined.

## Writing a Use Case Scenario

A Use Case scenario is a story that:

- Describes how an actor uses the system and how the system responds to the actions of the actor.

To say that a Use Case scenario is a story is a great metaphor. A story has all of the characteristics that were described about scenarios. Stories are a sequence of specific events that happen to specific people and things. In a Use Case scenario, the people are the actors and sometimes other people that the actor is interacting with. The events are the actions that take place between the actor and the system.

- Has a beginning, a middle, and an end.

Like any good story, a Use Case scenario has a beginning which tells us what the actor was doing when the use case begins. For a given use case, it is important that the beginning of each scenario is the same. This beginning text is often called the *trigger point* of the use case. This means that the use case has a specific starting point relative to the system. For example, in the Create a Reservation use case, the booking agent being requested to create a reservation will be the trigger.

The middle of the Use Case scenario provides the bulk of the details about how the actor interacts with the system and possibly other people or actors or other external systems.

The end of the Use Case scenario tells us how the story ends. Was it a happy ending? (Was the use case successful?) Or was it a sad ending? (Did the customer fail to reserve a room, because the hotel was booked up for that weekend?)

The following is an example a Primary (Happy) Use Case scenario for the Create a Reservation use case:

### In the Beginning

The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel.

The beginning of the Create Reservation use case has the booking agent receiving a request to make a reservation.



---

**Note** – We have avoided specifying the source of that request, but we could have stated it was a phone call from a customer.

---

## In the Middle

The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required. The booking agent then submits the entered details. The system finds rooms that will be available during the period of the reservation and allocates the required number and type of rooms from the available rooms. The system responds that the specified rooms are available, returns the provisional reservation number, and marks the reservation as “held”. The booking agent accepts the rooms offered.

This narrative is fairly terse and has no specific details of dates or room types. You may specify more details if you believe that the details are of significant value. For instance, you might want to specify boundary conditions for valid and invalid dates.

This narrative has also avoided using any specific user interface (UI). Therefore, it can be used for a variety of UIs. For example, the narrative will be applicable for a voice UI or online bookings, where the booking agent could be changed to Actor. However, there are many cases when you will have to specify the UI details. This is particularly true when discussing the scenarios with some of the users who prefer to discuss screens, buttons, lists, and so on.

The booking agent selects that the customer has visited one of the hotels in this group before, and enters the zip code and customer name. The system finds and returns a list of matching customers with full address details. The booking agent selects one of the customers as being the valid customer. The system assigns this customer to the reservation. The booking agent performs a payment guarantee check. This check is successful.

## In the End

The system assigns the payment guarantee to the reservation and changes the state of the reservation to “confirmed”. The system returns the reservation ID and booking details.

The end of the scenario describes the *post condition* of the use case. At this stage the system has completed its work and the booking agent only needs to inform the customer of the booking details.



---

**Note** – By delegating the payment guarantee to a sub-use case, we have avoided having to specify the details of the payment guarantee at this point.

---

The following is an example a Secondary (Sad) Use Case scenario for the Create a Reservation use case:

### In the Beginning

The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel.

### In the Middle

The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required. The booking agent then submits the entered details. The system responds that there are no rooms available of any type for date range specified in the request.

### In the End

The use case ends.

The following is another example of a Secondary (Sad) Use Case scenario for the Create a Reservation use case:

### In the Beginning

The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel.

### In the Middle

The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required. The booking agent then submits the entered details. The system responds that the

specified rooms are available, returns the provisional reservation number, and marks the reservation as “held”. The booking agent accepts the rooms offered.

The booking agent selects that the customer has visited one of the hotels in this group before, and enters the zip code and customer name. The system finds and returns a list of matching customers with full address details. The booking agent selects one of the customers as being the valid customer. The system assigns this customer to the reservation. The booking agent performs a payment guarantee check. The check is unsuccessful. The booking agent performs a second payment guarantee check. The check is unsuccessful. The booking agent cancels the request.

### In the End

The system removes the reservation and frees the allocated rooms.

There are obviously many more successful and unsuccessful scenarios that can be documented.

In addition, you should write scenarios for the different actors (roles) that will use this use case. In this example, you should write scenarios for an online booker and the Travel Agent System. If the scenarios are significantly different, you may consider creating a separate use case for each different actor (role).

## Supplementary Specifications

Some of the project information that you gather cannot be stored with the use cases because this information needs to be shared by several use cases.

---

**Note** – Some of the project information is often shared by two or more (often all) of the use cases.

---

This additional information can be documented in a Supplementary Specification Document, which often contains:

- NFRs
- Project Risks
- Project Constraints
- Glossary of Terms

Many companies create a Supplementary Specification Document to document this information. However, the name of this document often varies.

---

**Note** – Project risks and constraints will be discussed later in the course.

---

---

**Note** – A Supplementary Specification Document may have been created at the beginning of the project. In that case, you will just add new details to the document at this stage.

---

## Non-Functional Requirements (NFRs)

*Non-functional requirements* (NFRs) define the qualitative characteristics of the system. As in an animal, the NFRs describe strength, speed, and agility of the internal features of the animal. How fast can the animal move? How much weight can the animal carry?

- NFRs describe features of a system that support *how* an operation is performed.

NFRs include: how fast an operation is processed, how many operations can be performed simultaneously, how easy it is to add new features to the system, how easy the system is to manage, how easy the system is to use, and so on.

- Any adverbial phrase can be an NFR.

An adverb is the part of speech that expresses some relation of manner or quality, place, time, degree, number, cause, and so on. With respect to software systems, adverbs often express qualities of how well the system must perform.

NFR Examples:

- NFR1: The system must support 200 simultaneous users in the Web application.
- NFR2: The process for completing any reservation activity must take the average user no more than 10 minutes to finish.
- NFR3: The capacity of reservation records could grow to 2,600 per month.
- NFR4: The Web access shall use the HTTPS transport layer when critical customer information is being communicated.
- NFR5: The numerical accuracy of all financial calculations (for example, reports and customer receipts) should follow a 2-significant-digit precision with standard rounding of intermediate results.
- NFR6: The System must be available “7 by 24 by 365”. However, the applications can be shut down for maintenance once a week for one hour. This maintenance activity should be scheduled between 3 a.m. and 6 a.m.
- NFR7: Based on historical evidence, there are approximately 600 reservations per month per property.
- NFR8: The search for available rooms must take no longer than 30 seconds.

## Glossary of Terms

The Glossary of Terms defines business or IT terms that will be used in the project.

This is a living document, which should be appended with new terms, or amended if a term is found to be incorrect or needs redefinition.

Table 4-1 Shows a sample of terms that would be used in the Hotel System.

Term	Definition
Reservation	An allocation of a specific number of rooms, each of a specified <i>room type</i> , for a specified period of days.
Date Range	Specifies a start date and an end date.
Room Type	A room type indicates the number of beds, <i>basic rate</i> , and configuration of the room.
Room Number	A number that uniquely identifies a room within a <i>hotel</i> .
Room Name	Some rooms, such as conference rooms, are identified by a name instead of a number.
Room	A resource that can be allocated to a reservation, and is occupied by that reservation <i>customer</i> and their <i>guests</i> for the <i>date range</i> of the <i>reservation</i> . A room is identified by either a <i>room name</i> or a <i>room number</i> . Each room is assigned a <i>room type</i> .
Payment Guarantee	Debit/Credit card pre-authorization or purchase order from either corporate companies or travel agents.
Basic Rate	The per day price for a room type without any additional <i>in-line charges</i> or <i>promotions</i> .
Receipt	A document given to the <i>customer</i> at the time of <i>check out</i> . A receipt contains customer information and a summation of all of the charges incurred during the customer's stay at the property. Charges include room charge, taxes, and <i>line-item charges</i> , such as food and beverage charges and phone call charges.
Promotion	A discount or upgrade offered to the <i>customer</i> in hopes of selling a particular service or product. A discounted room rate for weekday customers is an example of a promotion used to get more business during days when there is customarily low <i>occupancy</i> .

**Table 4-1** Example Glossary of Terms for the Hotel System



# Creating a Use Case Form

## Description of a Use Case Form

A Use Case form provides a tool to record the detailed analysis of a single use case and all its scenarios. Table 4-2 lists the elements of the Use Case form.

**Table 4-2** Use Case Form Elements

Form Element	Description
Use Case Name	The name of the use case from the Use Case diagram.
Description	A one-line or two-line description of the purpose of the use case.
Actors	This element should list all relevant actors that are permitted to use this use case.
Priority	This is used to describe the relative priority of this use case. Priority is often in the form of MuSCoW prioritization, which is Must have, Should have, Could have, or Won't have.
Risk	A High, Medium, or Low ranking of this use case's risk factors.
Pre-conditions and assumptions	The conditions that must be true. If these conditions are not true, the outcome of the use case cannot be predicted.
Extension Points	A list of any extension points used by this use case.
Extends	A list of any use cases that this use case extends.
Trigger	The condition that "informs" the actor that the use case should be invoked.
Flow of Events	The primary trace of user actions and events that constitute this use case.
Alternate Flows	Any and all secondary traces of user actions and events that are possible in this use case.
Post-conditions	The conditions that shall exist after the use case has been completed.

**Table 4-2** Use Case Form Elements (Continued)

Form Element	Description
Business Rules	A list of business rules that must be complied with and that are related to this use case. These rules might be referred to during the execution of the use case in the main flow and the alternate flow, but this is not always necessary. You can describe these rules in this form. Alternatively, you can refer to the list in the Supplementary Specification Document.
Non-Functional Requirements	A list of the NFRs that are related to this use case. You can either summarize the NFRs or list their codes from the Supplementary Specification Document.
Notes	Any other information that can be of value regarding this use case.

Some methodologies recommend more or less analysis of the use cases. The Analysis workflow presented in this module tends to be more detailed. Less detailed analyses might only determine the Flow of Events. After learning about this Use Case form, you can choose which elements are important for your project.

Use Case forms are not standard. There are different styles that can be used to create a Use Case form. For example, some forms separate some items from the Alternate Flow section into an additional section named Exception Flow. The exact description of these differences in style vary between companies.

## Creating a Use Case Form

Perform these steps to determine the information for the Use Case form:

4. Determine a brief description from the primary scenarios.
5. Determine the actors who initiate and participate in this use case from the Use Case diagrams.
6. Determine the priority of this use case from discussions with the stakeholders.
7. Determine the risk from scenarios and from discussions with the stakeholders.
8. Determine the extension points from the Use Case diagrams.
9. Determine the pre-conditions from the scenarios.
10. Determine the trigger from the scenarios.

11. Determine the flow of events from the primary (happy) scenarios.
12. Determine the alternate flows from the secondary (sad) scenarios.
13. Determine the business rules from scenarios and from discussions with stakeholders.
14. Determine the post-conditions.
15. Determine the new NFRs from discussions with stakeholders.
16. Add notes for information—gathered from discussions with stakeholders—that does not fit into the standard sections of the form.

### Fill in Values for the Use Case Form

Fill in elements from the information already gathered or by reviewing these forms with the stakeholders to add or enhance the requirements. Table 4-3 is an example of some elements required for the Create Reservation use case.

**Table 4-3** Partial Example Use Case Form for the Create Reservation Use Case

Form Element	Description
Use Case Name	Create Reservation
Description	The Customer requests a reservation for hotel rooms for a date range. If all the requested rooms are available, the price is calculated and offered to the Customer. If details of the customer and a payment guarantee are provided, then the reservation will be confirmed to the Customer.
Actor(s)	Primary: Booking Agent, Online Booker, Travel Agent System Secondary: None Note: Primary actors are proxies for the Customer.
Priority	Must have Note: This use case is essential to this system.
Risk	High Note: The risk is high primarily because of the complexity of identifying if rooms are available and the number of different actor roles that can use this use case.

**Table 4-3** Partial Example Use Case Form for the Create Reservation Use Case

Form Element	Description
Trigger	A Customer wishes to make a reservation in the hotel.
Pre-conditions	At least one room exists in the hotel. Primary Actor can be identified.
Post-conditions	One reservation is added. Payment guarantee details are recorded.
Non-Functional Requirements	<i>NFR1 (Simultaneous Users)</i> <i>NFR2 (Duration of Use Case)</i> <i>NFR4 (Web Security)</i> <i>NFR6 (System Availability)</i> <i>NFR8 (Max Time for Room Availability Search)</i>
Notes	A fast method of checking room availability is still under investigation.

Primary actors are the actors who must perform the use case to satisfy their job roles. A secondary actor is any other actor who can participate in the use case, but cannot initiate it. For example, a booking agent is primarily responsible for creating reservations along with an online booker and the Travel Agent System. Therefore, these actors are the primary actors. However, the Electronic Payment System will just participate in the Perform Payment Transaction use case. Therefore, Electronic Payment System is a secondary actor.

**Note** – The example Hotel System assumes that the scenarios for the booking agent, online booker, and Travel Agent System are identical except for the interface to the device. If this is not the case, a minor variation can be shown by using the alternate paths. However, if there are major differences, a separate use case may be used.

You can document that other actors can assume the role of a booking agent. For example, the receptionist and duty manager roles can assume the role of a booking agent.



## Fill in Values for the Main Flow of Events

Table 4-4 is an example of the main flow of events of the Create Reservation use case. These are derived from a primary scenario.

**Table 4-4** Main Flow of Events of the Create Reservation Use Case

Main Flow of Events	<p>1: Use Case starts when Customer requests to create a reservation</p> <p>2: Customer enters types of rooms, arrival date, and departure date [A1] [A2]</p> <p>2.1: System creates a reservation and reserves rooms applying BR3 [A3]</p> <p>2.3: System calculates quoted price applying BR4</p> <p>2.3.1 System records quoted price</p> <p>2.4: System notifies Customer of reservation details (including rooms and price)</p> <p>3: Customer accepts rooms offered [A5]</p> <p>3.1: Extension Point (new customer) [A6]</p> <p>3.2: Extension Point (payment guarantee) [A7]</p> <p>3.3: System changes reservation status to “confirmed”</p> <p>3.4: System notifies Customer of confirmed reservation details</p> <p>4: Use case ends</p>
---------------------	--



**Note** – It is common to use a Dewey Decimal style of numbering for flow of events in Use Case forms as shown in the above example. In this style, a Use Case form usually start with an organized sequence and hierarchy of substeps—for example 1, 1.1, 1.2, 1.3, 1.3.1, 1.4, 2, 2.1, 2.2, and so on. During the modeling process, you often need to add or remove steps, which can be achieved without the need to renumber the existing steps. This can affect the sequential numbering, but is less prone to mistakes.

## Fill in Values for the Alternate Flow of Events

Table 4-5 is an example of the alternate flow of events of the Create Reservation use case. These alternate flow of events are derived from the secondary scenarios and any remaining primary scenarios.

- Perform a *difference analysis* between the scenario used for the main flow and each of the other scenarios (in turn).
- The alternate flows are the steps that are different between the scenario used for the main flow and each of the other scenarios.

**Table 4-5** Alternate Flows of the Create Reservation Use Case

Alternate Flows	<p>A1: Customer can enter duration instead of departure date, go to step 2.1 [A2]</p> <p>A2: Failed date check BR1. Notify error to Customer, go to step 2</p> <p>A3: Complying with BR2, System determines that required rooms are not available, System upgrades one or more room types, go to step 2.1[A4]</p> <p>A4: No further upgrades available. Notify message to Customer, go to step 2</p> <p>A5: Rooms offered are declined, go to step A9</p> <p>A6: Customer already exists, Customer enters customer name and zip code, System searches for matching customers, notifies Customer of matching customers, Customer selects correct customer details, go to step 3.2 [A8]</p> <p>A7: Payment guarantee fails. Notify message to Customer, go to step 3.2</p> <p>A8: Existing customer not found, go to step 3.1</p> <p>A9: Reservation not confirmed, reservation deleted, use case ends</p> <p><i>At any time:</i> Customer may cancel the use case, use case ends [A9]</p> <p><i>After use case inactivity of 10 minutes:</i> use case ends [A9]</p>
-----------------	--

## Fill in Values for the Business Rules

Table 4-5 is an example of the business rules of the Create Reservation use case.

**Table 4-6** Business Rules of the Create Reservation Use Case

Business Rules (BR)	<i>BR1:</i> The arrival date must not be before today's date, and departure date must be after arrival date <i>BR2:</i> Overbooking is not allowed <i>BR3:</i> Reservations with assigned rooms but no payment guarantee have a status of "held" <i>BR4:</i> The quoted price is the sum the base price of the room types after applying BR5 and BR6 <i>BR5:</i> Seasonal Adjustment can be applied if reservation dates are applicable <i>BR6:</i> Offer adjustments can be applied if reservation qualifies <i>BR7:</i> Reservations with "held" status can be deleted <i>BR8:</i> Reservations with a status of "confirmed" must be linked to a payment guarantee and a customer <i>BR9:</i> Reservation must not exist without being linked to at least one room
---------------------	--

## Summary

- A Use Case scenario is written to provide a detailed description of the activities involved in one instance of the use case.
- Use Case scenarios should provide as many different situations as possible so that the whole range of activities for that use case are documented.
- Use Case scenarios provide much detail about a use case. An analysis of this detail is recorded in the Use Case form.

The activities of a use case are distilled into Flow of Events portion of the Use Case form. Alternate flows are identified from unusual situations in one or more scenarios.



# Creating Activity Diagrams

---

## Objectives

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

- Identify the essential elements in an Activity diagram
- Model a Use Case flow of events using an Activity diagram

## Additional Resources

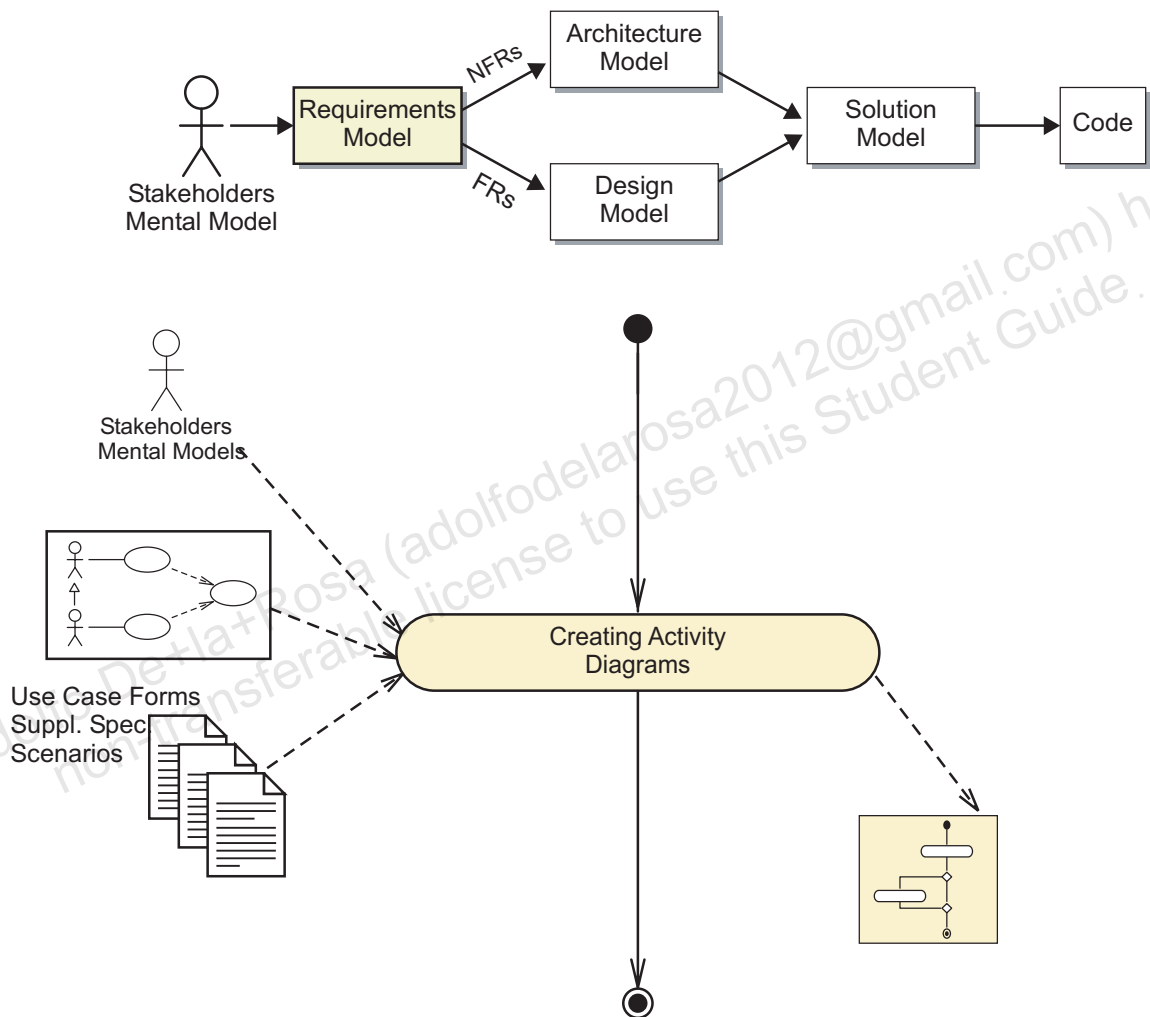


**Additional resources** – The following references provide additional information on the topics described in this module:

- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Harlow: Addison Wesley Longman, Inc., 1993.
- Rosenberg, Doug, Kendall Scott. *Use Case Driven Object Modeling with UML (A Practical Approach)*. Reading: Addison Wesley Longman, Inc., 1999.
- Rosenberg, Doug, Kendall Scott. *Applying Use Case Driven Object Modeling with UML (An Annotated e-Commerce Example)*. Reading: Addison Wesley Longman, Inc., 2001.
- The Object Management Group. “OMG Unified Modeling Language™ (OMG UML), Superstructure,”  
[<http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>],  
Version 2.2, February 2009.

# Process Map

This module covers the next step in the Requirements Analysis workflow: Creating Activity diagrams. Figure 5-1 highlights the activities and artifacts covered in this module.



**Figure 5-1** Creating Activity Diagrams Process Map

## Describing a Use Case With an Activity Diagram

Because use cases are essential to the success of a software project, you should verify that your mental model of the Use Case matches the stakeholder's mental model. You can verify a Use Case by creating a view of your mental model that illustrates the behavior of that Use Case. To do that, you can:

- Model the flow of events of the Use Case in an Activity diagram
- Validate the Use Case by reviewing the Activity diagram with the stakeholders

Activity Diagrams can also be used for the following:

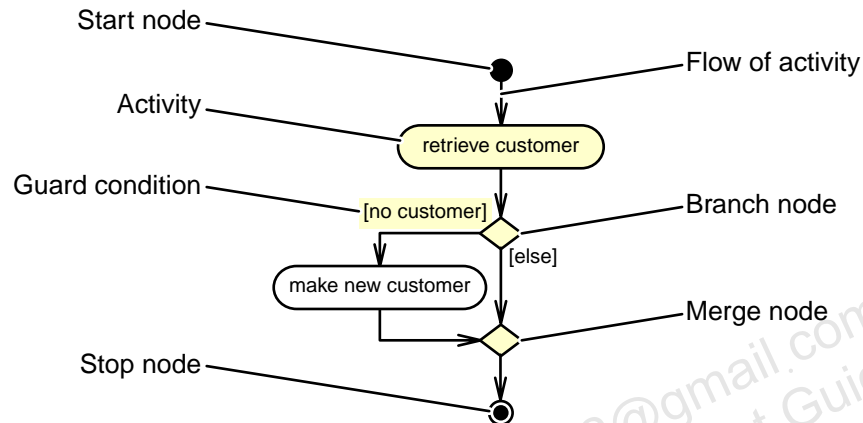
- Model Business Processes
- Model Detailed Design

Business process modeling can include multiple Use Cases and the interaction between actors (human and systems). Whereas a detailed design is often a small part of a Use Case depicted to a fine level of detail. For example, one activity in the analysis diagram may be linked to another diagram showing the internal low level sub activities.

To represent the Use Case in an Activity diagram, you must learn the essential features of these UML diagrams. The essential features of Activity diagrams is presented in the next section.

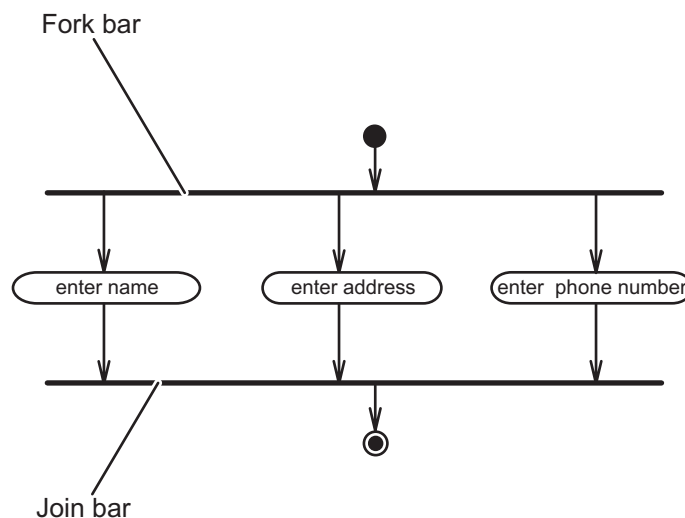
## Identifying the Elements of an Activity Diagram

An Activity diagram represents a sequence of actions (or activities) with a definite beginning and end-point. Figure 5-2 shows the essential elements of an Activity diagram.



**Figure 5-2** Example of an Activity Diagram

Activity diagrams can also represent simultaneous activities in which the activities can be performed in any order. Simultaneous activities are illustrated with fork and join bars. This notation can indicate a truly parallel activity as in the case of a multiprocessor system or it can indicate a set of activities with no specific order. Figure 5-3 illustrates an example for using fork and join bars.



**Figure 5-3** An Example of Concurrent Activities

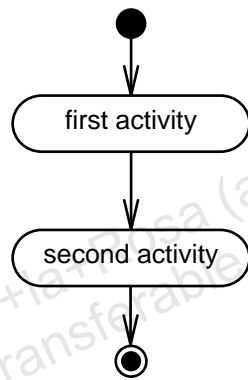
### Activities and Actions

Activities and actions are processes taken by the system or an actor.

- Activity nodes and action nodes use the same notation in UML
- An activity can be divided into other activities or actions
- An action is an activity node which cannot be divided within the context of the current view.
- A primitive form of action results in a change in the state of the system or the return of a value.

### Flow of Control

An Activity diagram must start with a Start node and end with a Stop node. Flow of control is indicated by the arrows that link the activities together. Figure 5-4 illustrates the flow of control.



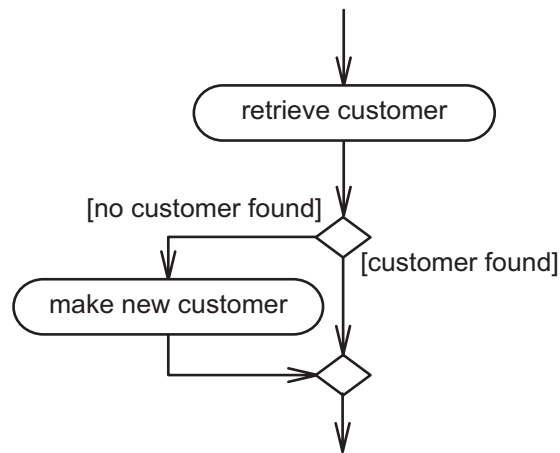
**Figure 5-4** An Activity Diagram Represents Flow of Control

### Branching

The branch and merge nodes represent conditional flows of activity:

- A branch node has two or more outflows, with Boolean predicates to indicate the selection condition.
- A merge node collapses conditional branches.

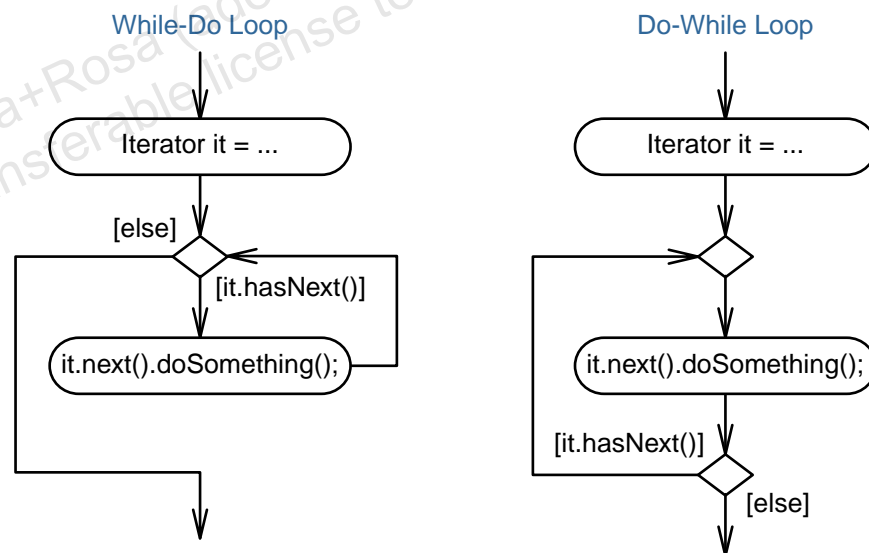
Figure 5-5 illustrates branch and merge nodes.



**Figure 5-5** Branch and Merge Nodes

## Iteration

You can achieve iteration by using branch nodes. Figure 5-6 shows two examples of iteration.

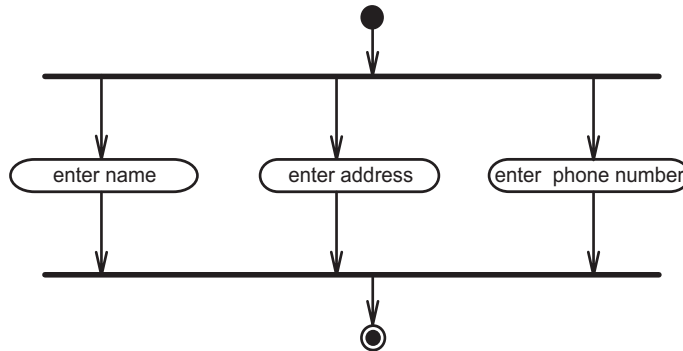


**Figure 5-6** Iteration Loops in an Activity Diagram

The first example (on the left) maps closely to a while-do loop in which the loop test is performed before the loop body. The second example (on the right) maps closely to a do-while loop in which the loop body is performed at least once before the loop test is performed.

## Concurrent Flow of Control

Use the fork and join bars to indicate concurrent flow of control. Figure 5-7 illustrates this.



**Figure 5-7** Concurrent Flow in an Activity Diagram

- Fork and join bars can represent either threaded activities or parallel user activities.

The fork and join bars can represent two distinct types of behavior:

- fork and join can represent threaded or truly concurrent activities within a computer system.
- fork and join can represent a set of activities that can happen in any, non-specified order.



## Passing an Object between Actions

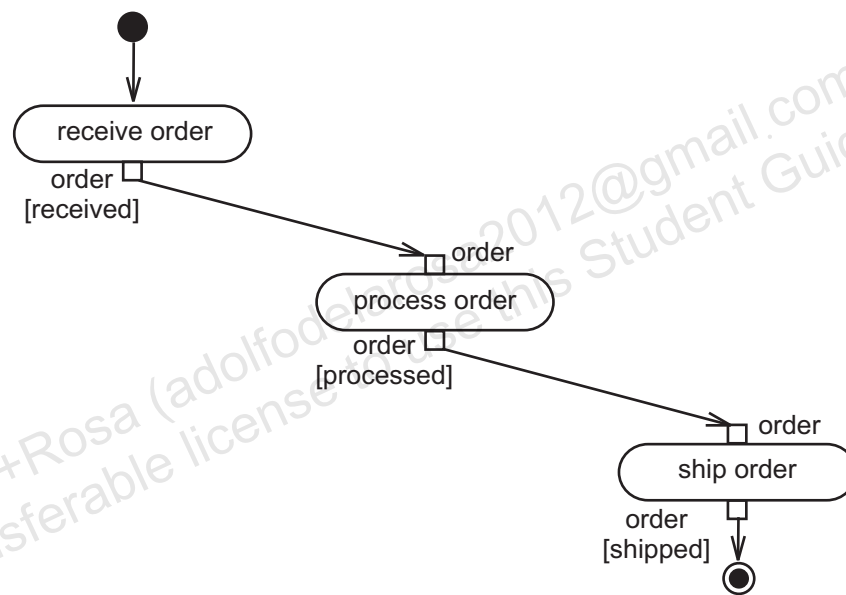
An Activity diagram can show objects being passed between actions

- A pin is a connection point of an action for object input or output.
- The name of the pin denotes the object being passed in or out.

A pin is shown as using a small rectangle attached above or below an activity node.

You can optionally show the state of the object that is being passed within square brackets, next to the object name.

Figure 5-8 illustrates the uses of pins.



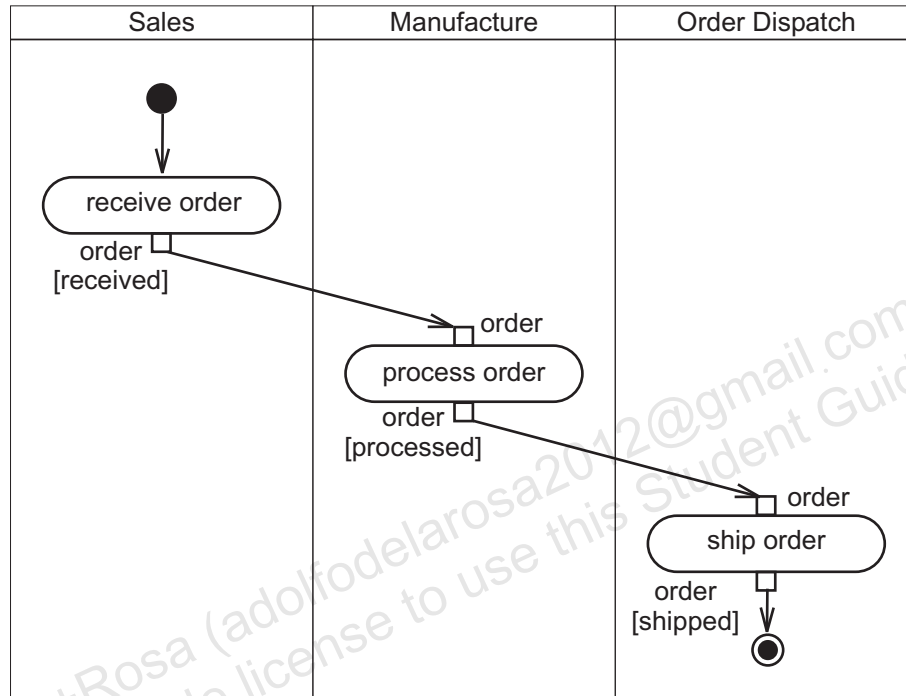
**Figure 5-8** Example of Object Flow between Activities

**Note** – In previous UML versions there were alternative styles used to show the passing of an object. These styles remain valid.



## Partitions in Activity Diagrams

An Activity diagram can show objects grouped into partitions (formerly called swimlanes). Partitions are labelled to represent the name of the group. Partitions can be either vertical, horizontal or both vertical and horizontal. Figure 5-9 illustrates an example of vertical partitions.



**Figure 5-9** Example of Vertical Partitions in an Activity Diagram

## Signals in Activity Diagrams

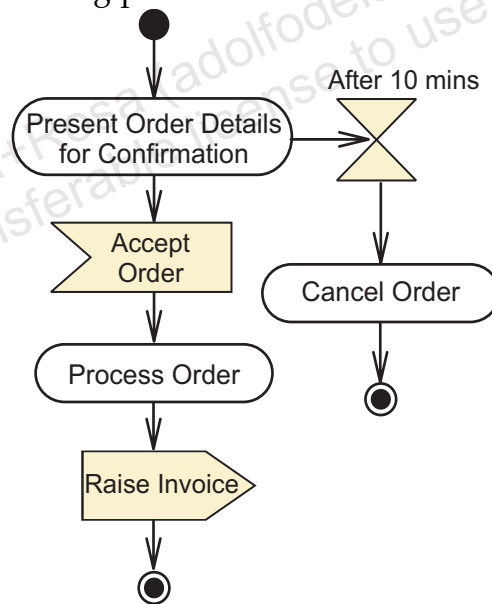
An Activity diagram can show the receiving of a signal, using an:

- Accept Event Action element
- Accept Time Event element

The source of the signal is often external to the current activity diagram. The Time Event is triggered at a specific time or after a time interval.

An Activity diagram can show the sending of a signal, using a Send Event Action element

The destination of the signal is often external to the current activity diagram. Figure 5-10 shows an example where Order Details are presented for confirmation. If the Accept Order signal is received within 10 minutes of the Order Details being presented, then the order is processed. When the Process Order activity is completed the Raise Invoice signal is sent. If the Accept Order signal is not received within 10 minutes of being presented then the Cancel Order activity will commence.



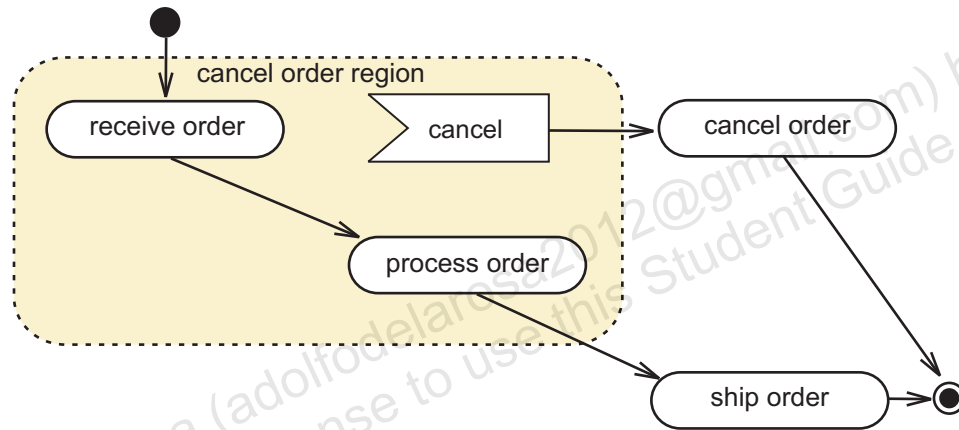
**Figure 5-10** Example of Send, Accept and Time Events.

## Interruptible Activity Regions

An Activity diagram can show a subset of activities that can be interrupted by an event.

On receipt of an Accept Event or Accept Time Event that occurs within the defined Interruptible Activity region, the current flow stops, and the flow continues from the interrupting event element.

Figure 5-11 shows an example where an order can be cancelled at any time until it is ready to be shipped.



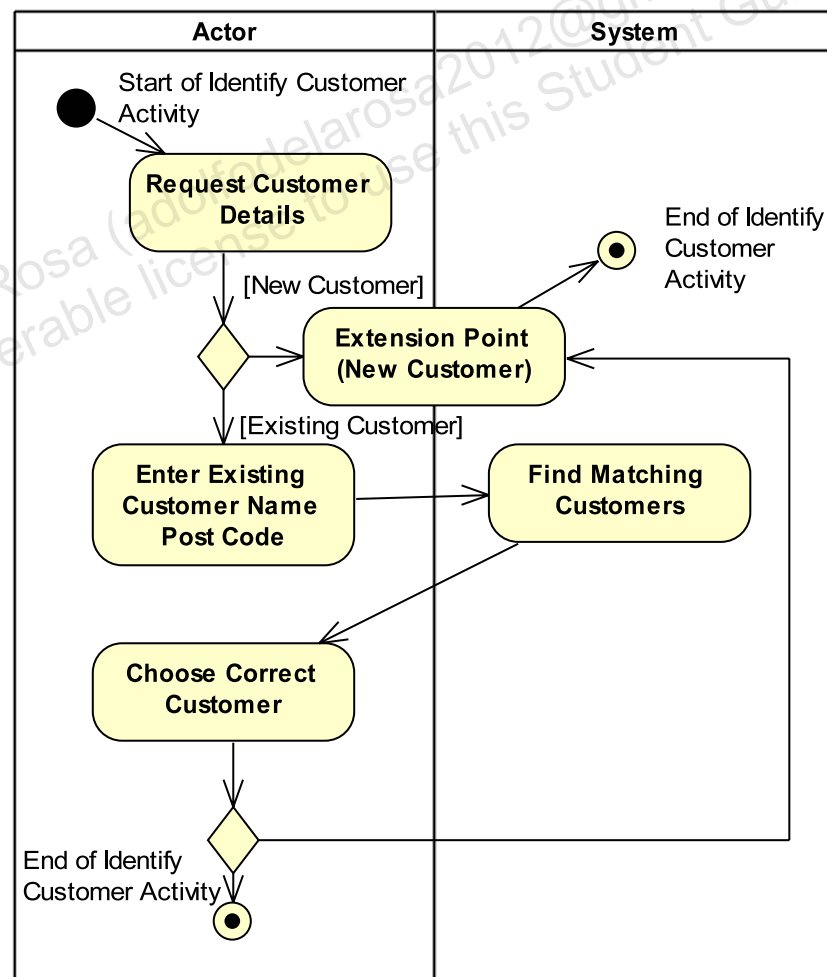
**Figure 5-11** Example of an Interruptible Activity Region

## Creating an Activity Diagram for a Use Case

To create an Activity diagram for a Use Case, analyze the flow of events field in the Use Case form:

- Identify activities
- Identify branching and looping
- Identify concurrent activities

Figure 5-12 illustrates a simple sequence of activities for part of the Create Reservation Use Case. This diagram shows the activities involved in identifying the customer, by either delegating the entry of the new customer details to the extension point (New Customer), or by the actor entering a subset of customer information in order to find the existing customer. If no existing customer is found then the extension point (New Customer) is used.



**Figure 5-12** Activity Diagram showing a subset of the Create Reservation Use Case

Figure 5-13 shows an Activity Diagram that closely represents the main flow and alternate flow paths of the Create Reservation Use Case Form.

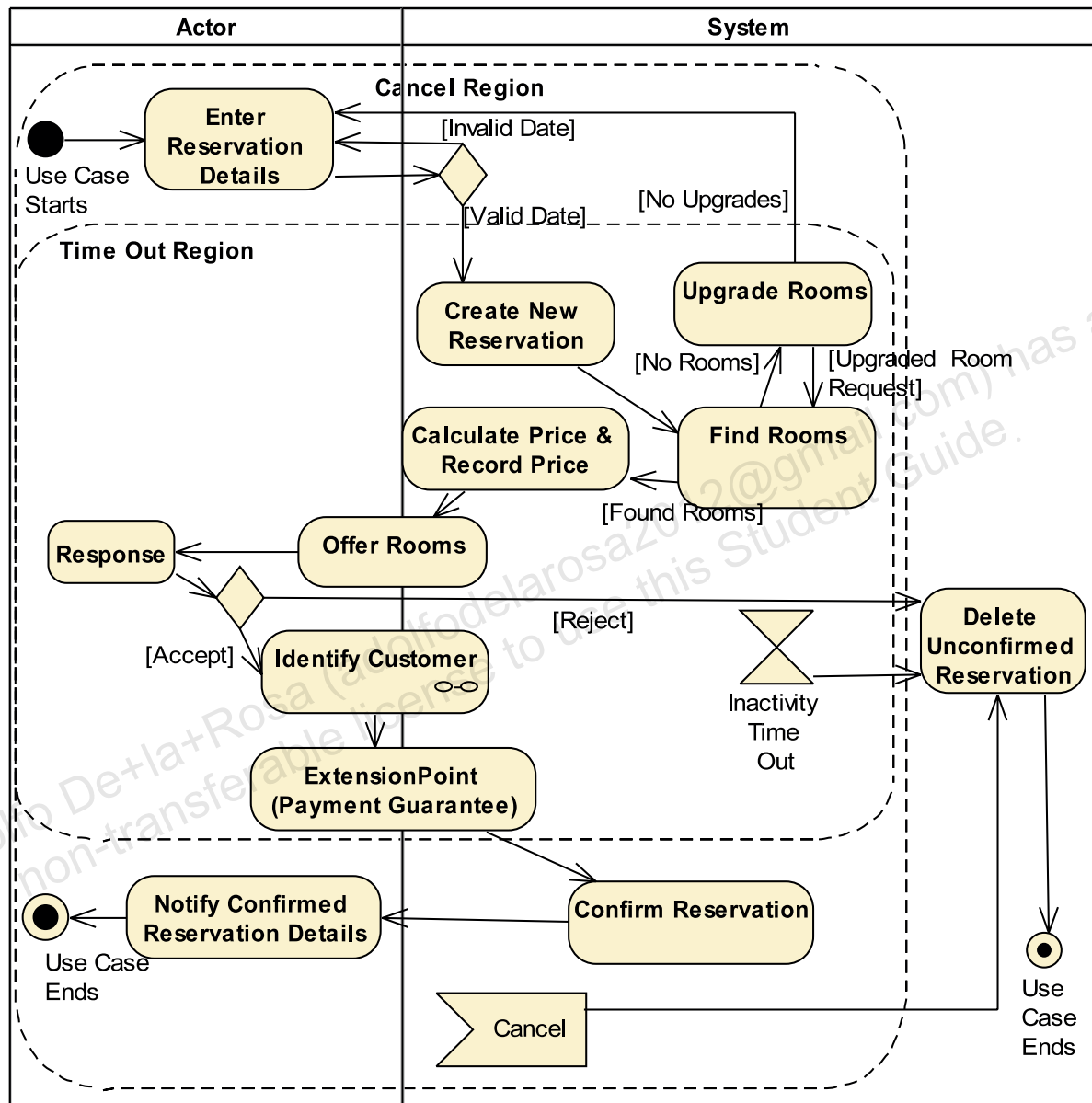


Figure 5-13 Activity Diagram for the Create Reservation Use Case

## Summary

This module described:

- The essential elements of an Activity diagram
- How you can visually represent the flow of events of a Use Case with an Activity diagram.  
You can present this view of the Use Case to the client-side stakeholders to verify that the development team understands the behavior of the Use Case.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.



# Determining the Key Abstractions

---

## Objectives

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

- Identify a set of candidate key abstractions
- Identify the key abstractions using CRC analysis

## Additional Resources

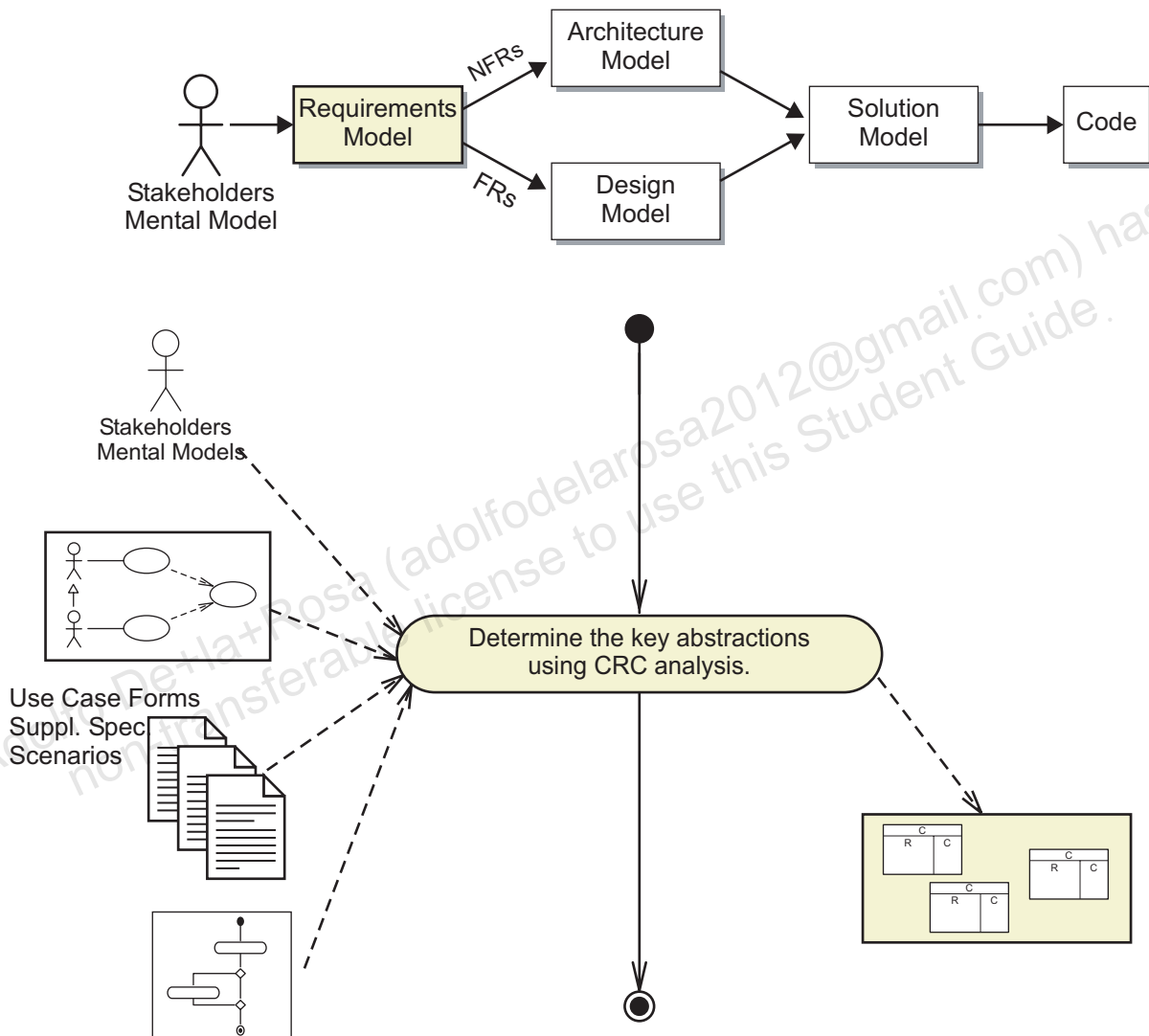


**Additional resources** – The following references provide additional information on the topics described in this module:

- Beck, Kent, Ward Cunningham. *A Laboratory For Teaching Object-Oriented Thinking*, [<http://c2.com/doc/oopsla89/paper.html>] 1989.
- Booch, Grady. *Object-Oriented Analysis and Design with Applications (2nd ed)*. The Benjamin/Cummins Publishing Company, Inc., Redwood City, 1994.
- Folwer, Martin, Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.

# Process Map

This module covers the next step in the Requirements Analysis workflow: discovering key abstractions. Figure 6-1 shows the activities and artifacts covered in this module.



**Figure 6-1** Key Abstractions Process Map

## Introducing Key Abstractions

“A key abstraction is a class or object that forms part of the vocabulary of the problem domain.” (Booch, page 162)

Key abstraction is the name for the primary objects within the system. These represent an integral part of the language in which the domain experts and the end users communicate about the business.

Determining the key abstractions for a domain is a process of discovery:

1. Identify all candidate key abstractions by listing all nouns from the project artifacts in a “Candidate Key Abstractions Form.”
2. Use CRC analysis to determine the essential set of key abstractions.

Key abstractions are recognized as objects that have responsibilities and are used by other objects (the collaborators).

This module describes only one technique for discovering key abstractions. There are other ways to manage this process, but it is beyond the scope of the course to cover all these techniques.

# Identifying Candidate Key Abstractions

Begin the process of identifying all of the unique nouns in the project artifacts by focusing on the following areas of the documents:

- The Main Flow and Alternate Flow sections of the use case forms.  
The nouns in these sections are usually essential to understand the problem domain. This is where you will find details and responsibilities of the problem domain.
- The other sections of the use case forms
- The use case scenarios
- The Glossary of Terms
- The Supplementary Specification Document.  
For large projects that use a non-iterative/incremental development process, this task would be time-consuming.  
For an iterative/incremental development process you will only need the current artifacts or changes to existing artifacts to be considered.

With practice you will be able to skip some of the nouns that are obviously not part of the domain.

## Identifying the Candidate Abstractions

An easy technique to scan the Use Case Form for nouns is to print a copy, and use a highlighter pen to mark each significant noun.

Here are a few excerpts from the Hotel System artifacts, with the nouns marked in **bold**:

- From the Create Reservation Use Case Form Description section:  
The **Customer** requests a **reservation** for **hotel rooms** for a **date range**. If all the requested **rooms** are available, the **price** is calculated and offered to the **Customer**. If **details of the customer** and a **payment guarantee** are provided, the **reservation** will be confirmed to the **Customer**.
- From the Create Reservation Use Case Form Main Flow section:  
1: **Use case** starts when **Customer** requests to create a **reservation**  
2: **Customer** enters **types of rooms**, **arrival date**, and **departure date** [A1] [A2]  
2.1: **Systems** creates a **reservation** and reserves **rooms** applying BR3 [A3]  
2.3: **System** calculates **quoted price** applying BR4  
2.3.1 **System** records **quoted price**  
2.4: **System** notifies **Customer** of **reservation details** (including **rooms** and **price**)  
3: **Customer** accepts **rooms** offered [A5]  
3.1: Extension Point (new **customer**) [A6]  
3.2: Extension Point (**payment guarantee**) [A7]  
3.3: **System** changes **reservation status** to "confirmed"  
3.4: **System** notifies **Customer** of **confirmed reservation details**  
4: Use case ends
- From the Create Reservation Use Case Form Alternate Flow section:  
A1: **Customer** can enter **duration** instead of **departure date**, go to step 2.1 [A2]  
A2: Failed date check BR1. Notify **error** to **Customer**, go to step 2  
A3: Complying with BR2, **System** determines that required **rooms** are not available, **System** upgrades one or more **room types**, go to step 2.1 [A4]  
A4: No further upgrades available. Notify **message** to **Customer**, go to step 2  
A5: **Rooms** offered are declined, go to step A9  
A6: **Customer** already exists, **Customer** enters **customer name** and **zip code**, **System** searches for matching **customers**, notifies

**Customer** of matching **customers**, **Customer** selects correct **customer**, go to step 3.2 [A8]

A7: **Payment guarantee** fails. Notify **message** to **Customer**, go to step 3.2

A8: Existing **customer** not found, go to step 3.1

A9: **Reservation** not confirmed, **reservation** deleted, use case ends

*At any time:* **Customer** may cancel the use case, use case ends [A9]

*After use case inactivity of 10 minutes:* use case ends [A9]

- From the Create Reservation Use Case Form Business Rules section:  
BR1: The **arrival date** must not be before **today's date**, and the **departure date** must be after the **arrival date**  
...  
BR3: **Reservations** with assigned **rooms** but no **payment guarantee** have a **status** of "held"  
...  
BR8: **Reservations** with a **status** of "confirmed" must be linked to a **payment guarantee** and a **customer**  
BR9: **Reservation** must not exist without being linked to at least one **room**
- From the Create Reservation Use Case Form Remaining Sections:  
...
- From the Supplementary Specification Documents. For example the Project Glossary, see Table 6-1

**Table 6-1** Project Glossary Terms

Term	Definition
<b>Reservation</b>	An allocation of a specific <b>number of rooms</b> , each of a specified <b>room type</b> , for a specified <b>period of days</b> .
<b>Date Range</b>	Specifies a <b>start date</b> and an <b>end date</b>

**Note** – We have deliberately omitted some text with important nouns. This will enable you to find them, as part of your lab.



## Candidate Key Abstractions Form

The form for recording candidate key abstractions uses three fields:

- **Candidate key abstraction**  
This field contains a noun discovered from the project analysis artifacts.
- **Reason for Elimination**  
This field is left blank if the candidate becomes a key abstraction. Otherwise, this field contains the reason why the candidate was rejected.
- **Selected name**  
Ultimately, a key abstraction will become some sort of software component, such as a class. This field contains the name of the class if this entry is selected as a key abstraction.

Use this form to record all potential (candidate) key abstractions from the set of nouns in the project analysis artifacts. You will list the nouns in the first column. During the CRC analysis (described in the “Discovering Key Abstractions Using CRC Analysis” on page 6-11) you will fill in the rest of the columns.

Table 6-2 shows an excerpt from the Candidate Key Abstractions Form for the Hotel System.

**Table 6-2** Initial Candidate Key Abstractions Form for the Hotel Reservation System

Candidate Key Abstraction	Reason for Elimination	Selected Component Name
Reservation		
Customer actor		
System		
Customer		
Room		
Date Range		
Price		
Customer Details		
Payment Guarantee		



**Table 6-2** Initial Candidate Key Abstractions Form for the Hotel Reservation System

Candidate Key Abstraction	Reason for Elimination	Selected Component Name
Room Type		
Arrival Date		
Departure Date		
Quoted Price		
Reservation Details		
Reservation Status		
Confirmed Reservation		
Duration		
Customer Name		
Customer Zip Code		
Today's Date		
Period of Days		

## Project Glossary

The process of identifying candidate key abstractions is also a good opportunity to verify that your project glossary is up-to-date.

- Verify that all domain-specific terms have been listed and defined.  
A candidate key abstraction is usually a domain-specific term. All of these should be listed in the glossary.
- Identify synonyms in the project glossary and select a primary term to use throughout the documentation and source code.

Synonyms sometimes appear in the candidate key abstractions list. For example, a booking agent might refer “reservation” as “booking.” Therefore, you should verify that both terms, “reservation” and “booking” appear in the glossary and that the definition of “booking” refers to the definition of “reservation.”

## Discovering Key Abstractions Using CRC Analysis

After you have a complete list of candidate key abstractions, you need to filter this list to determine the *essential set of key abstractions*. One technique is CRC (class-responsibility-collaboration) analysis.

To perform CRC analysis:

1. Select one candidate key abstraction.
2. Identify a use case in which this candidate is prominent.
3. Scan the use case forms and the use case scenarios that contain this noun to determine responsibilities and collaborators.
4. Scan the Glossary of Terms for all references to the noun.
5. Document this key abstraction with a CRC card.
6. Update Candidate Key Abstractions Form based on findings.

This process is iterative. You will start with one key abstraction candidate and then evaluate whether the candidate is a true key abstraction. When you have updated the form (Step 5), you will then select another candidate from the list and continue this process until all candidates in the form have been considered.

Each step in this process is described in the following sections.



**Note** – CRC analysis was invented by Ward Cunningham and Kent Beck in the late 80s to help teach object-oriented design. For more information about CRC analysis visit the URL

<http://c2.com/doc/oopsla89/paper.html>.

## Selecting a Key Abstraction Candidate

Selecting a good key abstraction candidate is largely intuition, but here are a few tactics:

- Ask a domain expert.

It is useful to have a domain expert in the meeting during CRC analysis. This person can immediately tell the analysis team if a candidate is likely to be a real key abstraction. Their mental model can quickly distinguish an object from an attribute, subtype name, or operation name.

If you do not have access to a domain expert, then the following tactic can be useful.

- Choose a candidate key abstraction that is used in a use case name.  
The use case names are usually written with a leading active verb (like “manage” or “create”) followed by a noun (like “reservation”). The nouns in the use case names tend to be key abstractions.
- Choose a candidate key abstraction that is used in a use case form.  
These will contain most of the candidate key abstractions.

For the Hotel System, the noun “reservation” is a good candidate because it appears many times in the following areas of the project analysis:

- In these use case names:
  - Create *Reservation*
  - Update *Reservation*
  - Delete *Reservation*
- In many places throughout the use case forms. For example the Check In Customer use case form will describe assigning a Bill (Folio) to a *reservation*.

### Identifying a Relevant Use Case

To determine whether the candidate key abstraction is a real key abstraction, you must determine if the candidate has any responsibilities and collaborators. To find the responsibilities and collaborators, you need to scan the text of the use case, details FRs for the use case, and the scenarios for the use case.

To identify a relevant use case that might declare a candidate’s responsibilities and collaborators:

1. Scan the use case names for the candidate key abstraction.  
If the candidate key abstraction is mentioned in the name of a use case, it is very likely that the use case will be relevant.
2. Scan the use case forms for the candidate key abstraction.  
Likewise, if the candidate key abstraction is mentioned in the description of a use case, it is very likely that the use case will be relevant.
3. Scan the use case scenarios for the candidate key abstraction.

4. Scan the text of the use case scenarios to see if the candidate key abstraction is mentioned. If it is, the scenario will be relevant.

For the Hotel Reservation System, the noun “reservation” occurs in names of three use cases:

- Create *Reservation*
- Update *Reservation*
- Delete *Reservation*

These relevant use cases will be used to perform the next step in the CRC process – to discover the responsibilities and collaborations of the selected candidate key abstraction.

## Determining Responsibilities and Collaborators

Scan the scenarios and use case forms of the identified use cases for responsibilities and collaborators of the candidate key abstraction. The responsibilities of a key abstraction are any attributes, operations, or specifications of the range of data value for attributes (for example, the names of the values of a state variable). The collaborators are other objects (usually another key abstraction) with which the candidate key abstraction is associated.

If you cannot find any responsibilities, then you can reject this candidate. Use the second column of the Candidate Key Abstractions Form to record the reason that the candidate was eliminated.

Also, if you determine that a responsibility (usually an attribute name or the name of a subtype) is also on the candidate list, then you can reject that noun from the list.

For example, the artifacts for the Hotel System includes the following details that specify a few of the responsibilities and collaborators of the Reservation key abstraction:

- Glossary Term Reservation has the following definition: an allocation of a specific number of rooms, each of a specified room type, for a specified period of days.

This glossary entry specifies that a reservation contains two important data items, the arrival and departure dates. These are responsibilities in the form of attributes. The glossary entry also specifies that a reservation is related to one or more rooms. This relationship is considered a collaboration.

- Business Rule BR9: Reservation must not exist without being linked to at least one room.

This business rule entry specifies that a reservation must be linked to at least one room. This relationship confirms the collaboration between the Reservation and a Room.

- Business Rule BR8: Reservations with a status of “confirmed” must be linked to a payment guarantee and a customer

This business rule entry specifies that a reservation contains another important data item: the status of the reservation. This is a responsibility in the form of an attribute. The business rule also specifies that a reservation is linked to a payment guarantee and a customer. These relationships are considered as collaborations.

- Main Flow 3.3: System changes reservation status to “confirmed”

This main flow entry specifies confirms that a reservation contains the important data item: the status of the reservation. This is a responsibility in the form of an attribute.

Note that the analysis for the Reservation key abstraction has identified attributes (for example, arrival date) and collaborators. The attributes might also be on the candidate key abstractions list; these can be eliminated with the reason that they are attributes.

## Documenting a Key Abstraction Using a CRC Card

After a key abstraction has been identified, you will create a CRC card to record the responsibilities and collaborators of this key abstraction. Figure 6-2 shows a template of a CRC card.

Class Name	
Responsibilities	Collaborators

**Figure 6-2** A CRC Card Template

The class name field at the top of the CRC card is where you place the name of the key abstraction. Finding a good name is important, especially if there are synonyms for this concept. As a recommendation, keep the name simple and short, but do not use abbreviations unless the full name is too long.

Use the responsibility column to record the operations and attributes of the key abstraction. Use the collaborators column to record any relationships and behavioral collaborations for the key abstraction.

Figure 6-3 shows an example CRC card for the Reservation key abstraction.

Reservation	
Responsibilities	Collaborators
Reserves a Room  status (New, Held, Confirmed) arrival date departure date	Room Customer Payment Guarantee

**Figure 6-3** The CRC Card for the Reservation Key Abstraction

These cards are more than just a source of documentation. Use them as *role-playing exercises* that help the analysis team and the domain expert explore how each key abstraction is used by the system to accomplish the use cases. It is this role-playing of the use case scenarios that is essential to the CRC process.

For example, consider the Create Reservation use case scenario 1. One member of the analysis team would represent the reservation object (holding the Reservation CRC card) and another member of the team would represent the customer object (holding the Customer CRC card). A third member of the team would represent the Booking Agent actor. This person would interact with the two other players as the scenario specifies activities designated for the specific object; for example, adding the customer to the reservation.

The role-playing exercise is an informal process, but it can be a powerful technique for building a deeper understanding of the problem domain.

## Updating the Candidate Key Abstractions Form

If the candidate you selected has responsibilities, then enter the name of the key abstraction (from the CRC card) into the “Selected Name” field.



If the candidate you selected has responsibilities and no collaborators, then this candidate must be rejected. Enter an explanation of why the candidate was not selected as a key abstraction. Some reasons for rejecting a candidate include:

- Name of an attribute
- Name of a subtype
- Name of an external system or any actor
- Name of a value of an attribute (such as for a status attribute)

After several iterations of the CRC process, several key abstractions are discovered for the Hotel System. Many candidates were rejected. Table 6-3 shows an excerpt of the final Candidate Key Abstractions Form.

**Table 6-3** Final Candidate Key Abstractions Form for the Hotel Reservation System

Candidate Key Abstraction	Reason for Elimination	Selected Name
Reservation		Reservation
Customer actor	External to system	
System	The whole system	
Customer		Customer
Room		Room
Date Range	Synonym for Arr. and Dept. Date	
Price	Synonym for Quoted Price	
Customer Details	Same as Customer	
Payment Guarantee		Payment Guarantee
Room Type		RoomType
Arrival Date	Attribute of Reservation	
Departure Date	Attribute of Reservation	
Quoted Price	Attribute of Reservation	
Reservation Details	Same as Reservation	
Reservation Status	Attribute of Reservation	
Confirmed Reservation	Type of Reservation	

**Table 6-3** Final Candidate Key Abstractions Form for the Hotel Reservation System

Candidate Key Abstraction	Reason for Elimination	Selected Name
Duration	Derived from Arr. and Dept. Date	
Customer Name	Attribute of Customer	
Customer Zip Code	Attribute of Customer	
Today's Date	External to System	
Period of Days	Synonym for Duration	

Adolfo De-la-Rosa (adolfoelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

## Summary

In this module, you were introduced to the process of discovering the Key Abstraction of the problem domain. Here are a few important concepts:

- Key abstractions are the essential nouns in the language of the problem domain.
- To identify the key abstractions:
  - a. List all (problem domain) nouns from the project analysis artifacts in a Candidate Key Abstractions Form.
  - b. Use CRC analysis to identify the key abstractions (a class with responsibilities and collaborators) from the candidate key abstractions list.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Constructing the Problem Domain Model

---

## Objectives

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

- Identify the essential elements in a UML Class diagram
- Construct a Domain model using a Class diagram
- Identify the essential elements in a UML Object diagram
- Validate the Domain model with one or more Object diagrams

## Additional Resources

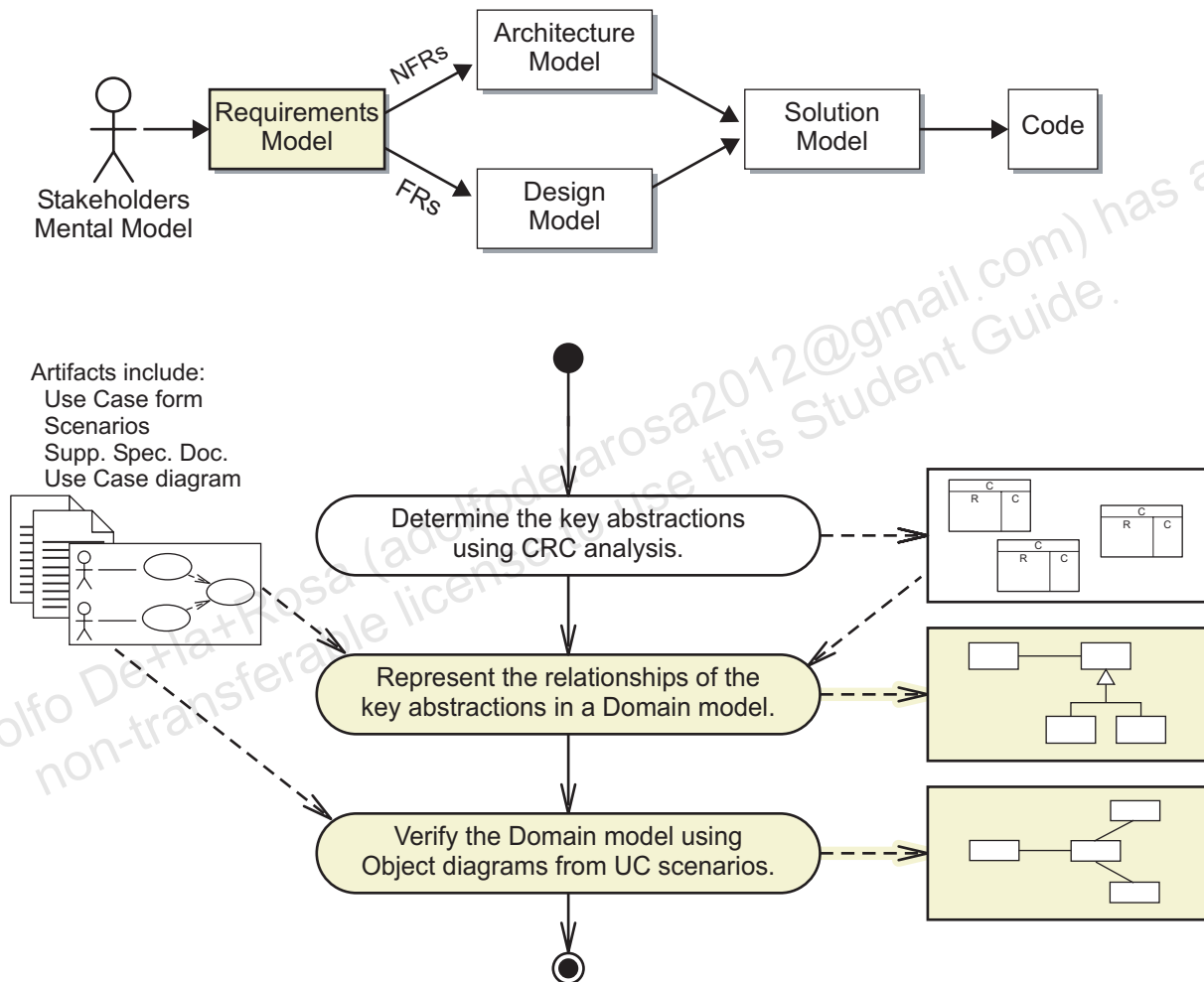


**Additional resources** – The following references provide additional information on the topics described in this module:

- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Booch, Grady. *Object Solutions (Managing the Object-Oriented Project)*. Reading: Addison Wesley Longman, Inc., 1994.
- Folwer, Martin, Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999
- The Object Management Group. “Unified Modeling Language (UML), Version 2.2”  
[<http://www.omg.org/technology/documents/formal/uml.htm>].
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004.

# Process Map

This module covers the next step in the Requirements Analysis workflow: creating the Domain model from the key abstractions. Figure 7-1 shows the activities and artifacts covered in this module.



**Figure 7-1** Domain Model Process Map

## Introducing the Domain Model

“The sea of classes in a system that serves to capture the vocabulary of the problem space; also known as a conceptual model.” (Booch Object Solution page 304)

The *Domain model* is one of three major views of the Requirements model; the others are the Use Case model and the Supplementary Specification Document.

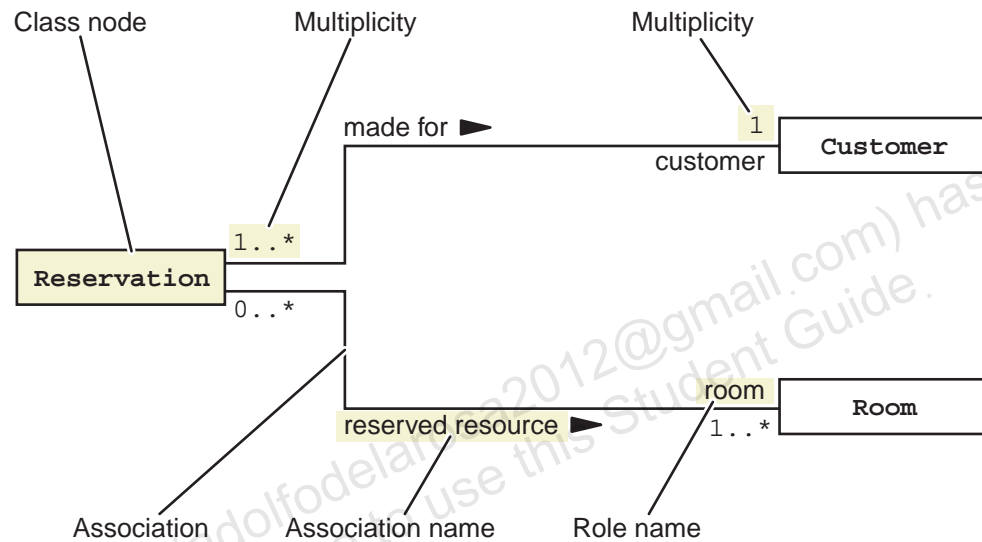
- The classes in the Domain model are the system’s key abstractions.  
The Domain model is a visual representation (a UML Class diagram) of the key abstractions that are discovered during analysis. Each key abstraction becomes a class in the Class diagram.
- The Domain model shows relationships (collaborators) between the key abstractions.  
CRC analysis identified the classes (key abstractions), responsibilities, and collaborations between classes. All of these features can be represented in the Domain model. It is the Domain model, not the CRC cards, that is a long-term artifact of the project.

This module describes the Domain model, how it is constructed, and the UML diagrams that represent it.



## Identifying the Elements of a Class Diagram

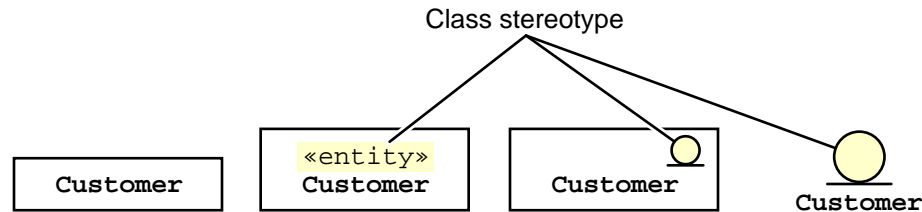
The *Class diagram* in the UML is probably the most recognized. It is used to visually represent classes, the members of the class, and the relationships between classes (usually called associations). Figure 7-2 shows an example Class diagram with a few of the features labelled by callouts.



**Figure 7-2** Elements of a UML Class Diagram

## Class Nodes

Class nodes represent classes of objects within the model. Class nodes can take many visual forms in UML, however, a class node is represented by a rectangle with the name of the class in a bold, monospaced font. Several different representations of the `Customer` class is shown in Figure 7-3.



**Figure 7-3** Different Ways to Represent a Class Node

A class node can represent:

- Conceptual entities, such as key abstractions  
During the Analysis workflow, class nodes in the Domain model represent key abstractions. These are conceptual entities that do not correspond to software components.
- Real software components  
During the Design workflow, class nodes usually represent a physical software component. The component is usually a class, but can also be other components. For example, a class node might represent an EJB technology entity bean.

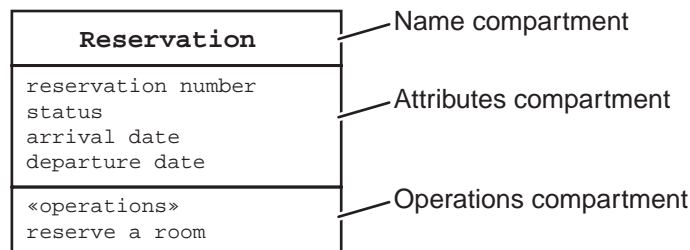
A stereotype can help identify the type of the class node. A stereotype in UML can take one of two forms. The textual form uses a word or phrase surrounded by guillemet characters; for example, «Entity». A stereotype can also take an iconic form; for example, an Entity can also be symbolized by a circle with a line under it.

## Class Node Compartments

A class node may be split into three or more compartments:

- The name compartment records the name of the class.  
This compartment is always the top of the node. The stereotype of the class must also be placed in the Name compartment.
- The attributes compartment records attributes of the class.  
This compartment is placed below the Name compartment. The attributes can be conceptual (see Figure 7-4) or detailed. You will see examples of detailed attributes later in the course.
- The operations compartment records operations of the class.  
This compartment is placed below the Attributes compartment. Methods can be conceptual or actual method signatures. You will see examples of method signatures later in the course.
- Additional compartments may be added.  
These compartments show additional information. These compartments are placed below the Methods compartment. You can use these compartments for any purpose, such as recording exceptions thrown by any of the methods or recording a textual description of the responsibilities of the class.

Figure 7-4 shows a class node for the Reservation class with three compartments visible.



**Figure 7-4** An Example Class Node With Compartments

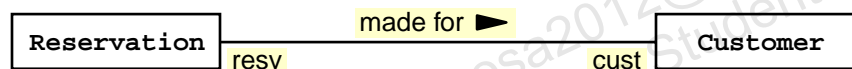
## Associations

*Associations* represent relationships between classes. Associations are manifested at runtime in which two objects are associated with each other, usually with an object reference.

It is important to understand that Class diagrams represent static information about the relationships between classes. A Class diagram *does not* represent the dynamic relationships of objects at runtime. It is better to think of a Class diagram as a set of constraints on the dynamic, runtime nature of objects.

### Relationship and Roles

Figure 7-5 shows an example association. This association would be read as “A reservation is made for a customer.”



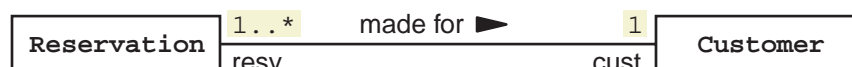
**Figure 7-5** Example Association With Relationship and Role Labels

The direction arrow indicates which direction to read the association. Role names indicate what role that a particular class is taking in the association. In Figure 7-5, customer is the role of the association between Reservation and Customer. However, if the role name is the same as the class name, then the role name can be eliminated. In this example, both role names could be eliminated.

### Multiplicity

Multiplicity determines how many objects might participate in the relationship.

Figure 7-6 shows an example association with multiplicity labels. This association would be read as “A reservation is made for one and only one customer.” Reading it in the other direction is “A customer can make one or more reservations.”



**Figure 7-6** Example Association With Multiplicity Labels

The default multiplicity for any side of the association is one. However, it is a good practice to explicitly show the “one” multiplicity labels. The multiplicity label can take several syntactic forms. These are listed in Table 7-1.

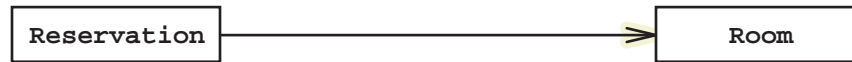
**Table 7-1** Different Types of Multiplicity

Multiplicity Syntax	Meaning
1	One and only one object.
n..m	At least n objects, but no more than m. This is an inclusive range.
0..1	Zero or one object. This multiplicity value is used to represent that an association to an object is optional.
n,m	There are either exactly n objects or exactly m objects. This syntax can be used in combinations with the double-dot notation. For example, 1..2,4,6..10 is valid syntax.
0..* or *	There may be zero or more objects.
1..*	There is at least one object, but there might be more.

## Navigation

Navigation arrows on the association determine what direction an association can be traversed at runtime. This additional information is most important during design, but it can also be useful in analysis. An association without navigation arrows means that you can navigate from one object to the other and the other way around as well. For example, Figure 7-6 on page 7-8 shows the association implies that from a Reservation object you can retrieve (navigate to) the Customer object. It also implies that from a Customer object you can retrieve the set of reservations for that customer.

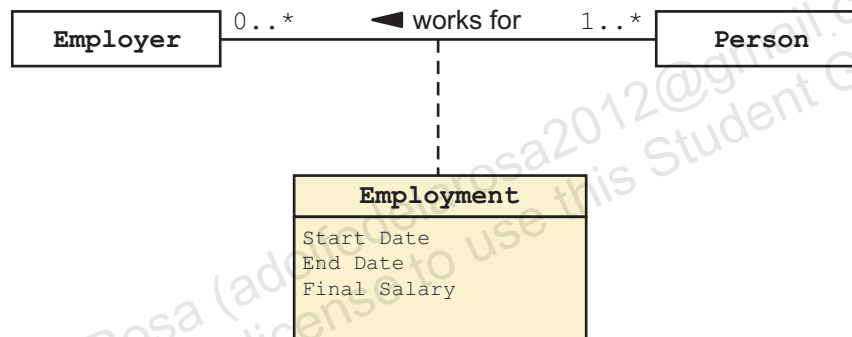
This is very flexible, but in some situations it might not be meaningful to the problem domain. For example, the functional requirements of the Hotel Reservation System do not require navigation from the room to the set of reservations. This constraint is illustrated in Figure 7-7. This association would be read as “From a reservation the system *can directly* retrieve the room, and from a room the system *cannot directly* retrieve the reservation for that room.”



**Figure 7-7** Example Association With Navigation Arrows

### Association Classes

Sometimes information is included in the association between two classes. For example, a person works for many employers. If you need to record details about each term of employment, this information belongs to the association between the person and the employer. You can show this using an association class. Figure 7-8 shows this.



**Figure 7-8** Example Association Class

**Note** – Association classes are used to specify the need to record the attributes shown in the association class. In the Design workflow, the designer will decide the best class to record these attributes.



# Creating a Domain Model

Starting with the key abstractions, you can create a Domain model using these steps:

1. Draw a class node for each key abstraction, and:
  - a. List known attributes.
  - b. List known operations.
2. Draw associations between collaborating classes.
3. Identify and document relationship and role names.
4. Identify and document association multiplicity.
5. Optionally, identify and document association navigation.

The following figures demonstrate the major steps of creating a Domain model from the set of key abstractions discovered in the previous analysis activity.



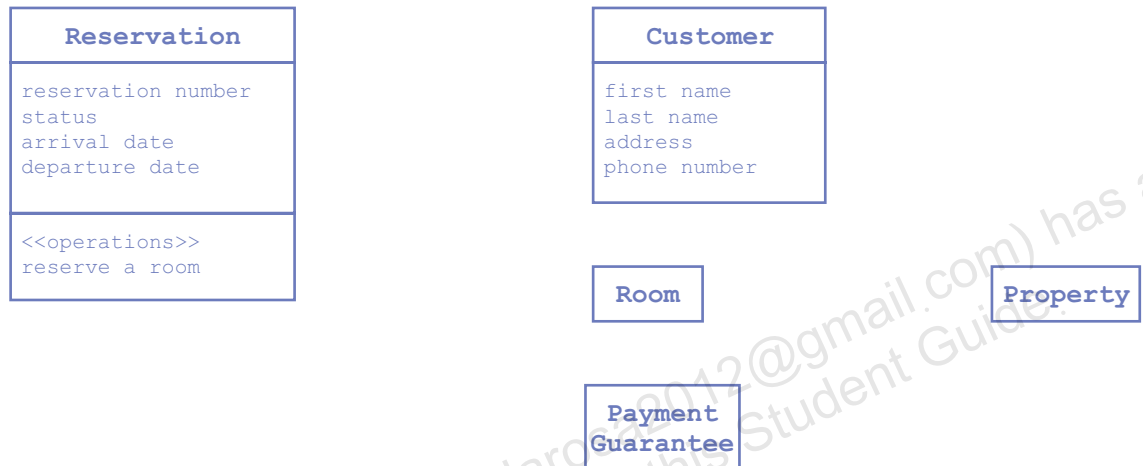
---

**Note** – The example used includes only a subset of classes. The example uses business rules that a hotel must have at least one room and a reservation must have at least one room. In addition, room names are used instead of room numbers.

---

## Step 1 – Draw the Class Nodes

The first step is to draw the class nodes for each key abstraction. Show all attributes and responsibilities, if you want more detail in the Domain model. Figure 7-9 illustrates this.



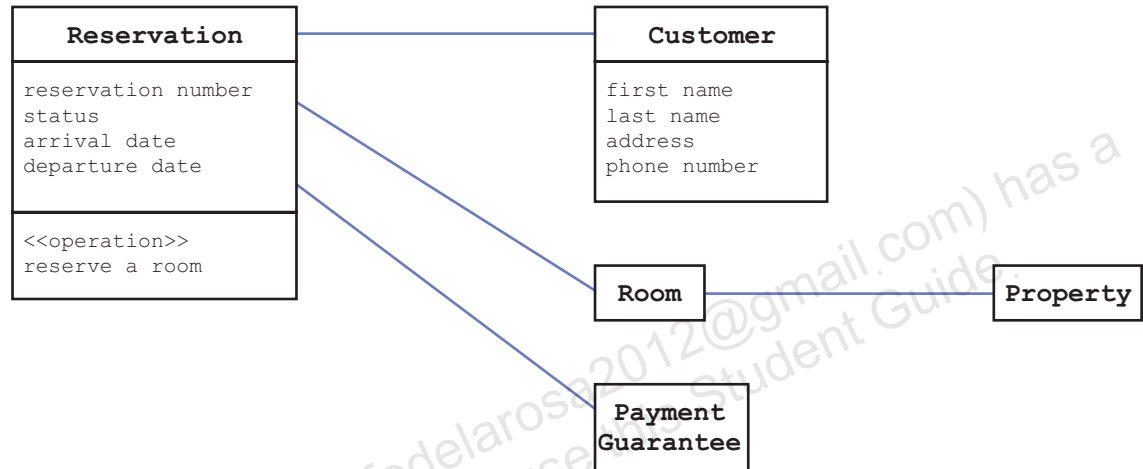
**Figure 7-9** Step 1 – Draw the Class Nodes

These diagrams have been drawn with precise orientation of each class node to facilitate showing a progression of steps where the class nodes stay in place. In practice, you should start with one class node (usually the most prominent key abstraction) and place it in the center of your drawing space. Add other class nodes around it as you draw the associations. As the diagram expands you will discover that you need to associate two classes that end up on different sides of the diagram. Because of this issue, you will have to draw the Domain model several times before you find the right visual structure.



## Step 2 – Draw the Associations

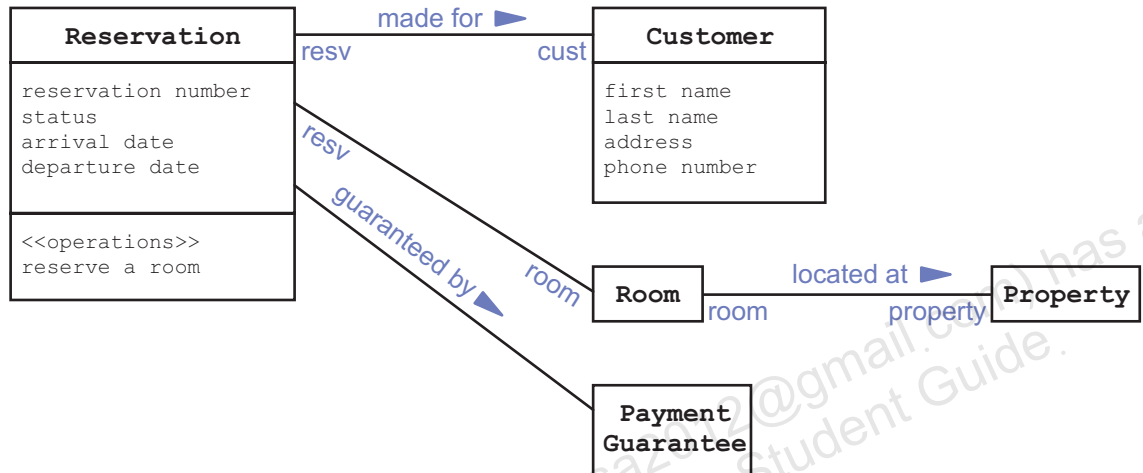
The next step is to draw association lines between each collaborating class. This information comes from the Collaboration column of the CRC card for each class. Figure 7-10 shows the associations for the key abstractions of the Hotel Reservation System.



**Figure 7-10** Step 2 – Draw Associations Between Collaborating Classes

## Step 3 – Label the Association and Role Names

The next step is to label the associations with association names and role names. Figure 7-11 illustrates this.

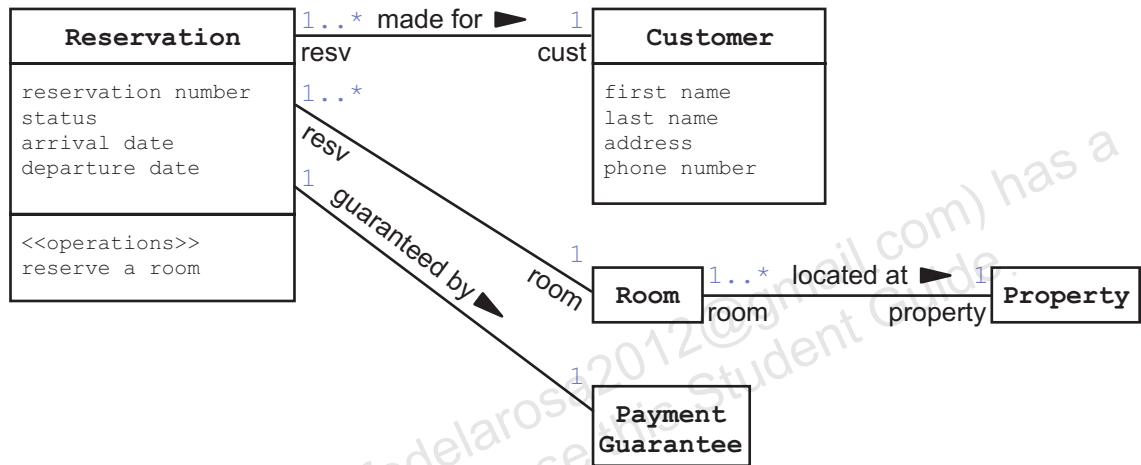


**Figure 7-11** Step 3 – Record the Relationship and Role Names

Association and role names tend to be obvious, so many analysts ignore this step.

## Step 4 – Label the Association Multiplicity

The next step is to add the multiplicity labels to the associations. The multiplicity information is usually not identified during CRC analysis. This information can be identified from the Use Case scenarios, the Use Case form, or a domain expert. Figure 7-12 illustrates this step.

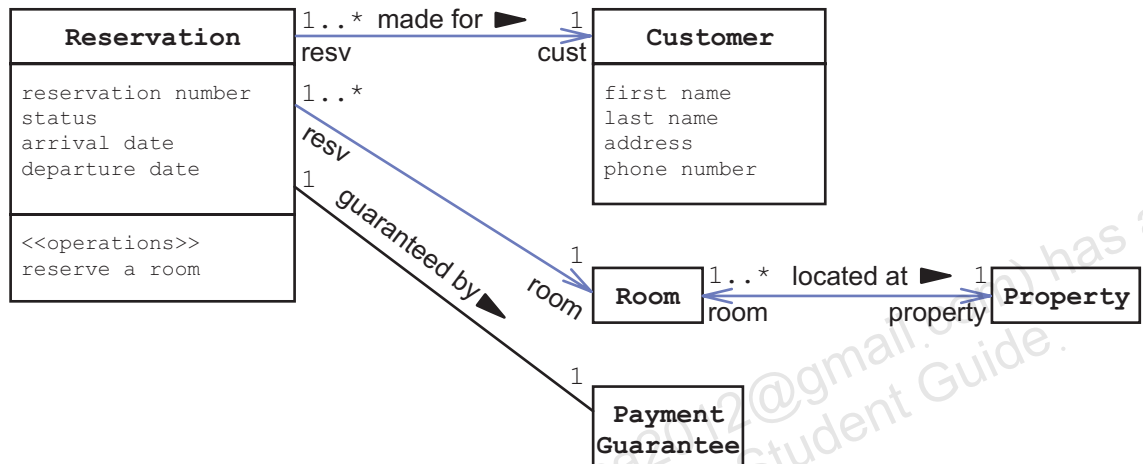


**Figure 7-12** Step 4 – Record the Association Multiplicity

It is common to make mistakes about the multiplicity of associations during analysis. In “Validating the Domain Model Using Object Diagrams” on page 7-20, you will learn a technique for identifying these problems.

## Step 5 – Draw the Navigation Arrows

The next step is to draw the navigation arrows on the associations, if they are known during analysis. Figure 7-13 illustrates this.



**Figure 7-13** Step 5 – Record the Association Navigation

This is done only when an FR constrains the Domain model or when the business analysts decides that the navigation of a particular association is meaningless for the domain. For example, it is not necessary to be able to navigate from a Room object to a reservation, so this association has a navigation arrow pointing from the Reservation to the Room class, but not in the other direction.

Notice that the association between Room and Property has a navigation arrow on both ends. This makes that association explicitly bidirectional. Associations without navigation arrows are implicitly bidirectional.

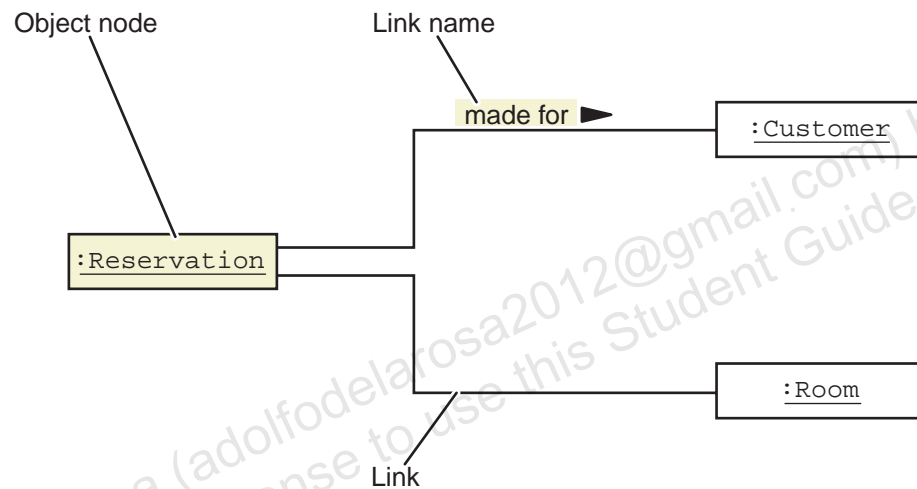
## Validating the Domain Model

In this section, you have seen how to construct a Domain model. How do you know that this diagram accurately models the problem domain? One technique is to build Object diagrams from the Use Case scenarios and check that the Object diagrams fit the Class diagram. This technique will be discussed in “Validating the Domain Model Using Object Diagrams” on page 7-20, but first this module describes the UML Object diagram syntax.

## Identifying the Elements of an Object Diagram

“A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time.” (UML v1.4 page 3-35)

The *Object diagram* in the UML visually represents objects and their runtime links. Figure 7-14 shows an example Object diagram with a few of the features labelled by callouts.



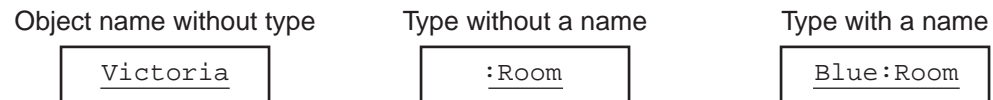
**Figure 7-14** Elements of a UML Object Diagram

**Note** – Link names are usually ignored in Object diagrams.



## Object Nodes

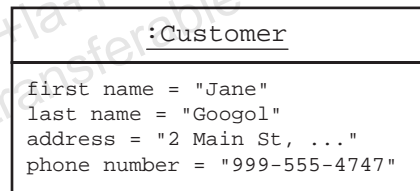
An object node usually includes some form of name and data type. Figure 7-15 shows this.



**Figure 7-15** Object Node Types

The first node, *Victoria*, does not include a data type. This means that you have an object, but you do not yet know what type of the object it is. This situation is rare. The second node shows an object with a type, but no name. This is very common. The third node shown an object with both a name and a data type. The name usually refers to a programming variable that refers to the object; also, the name could represent a name attribute of the object.

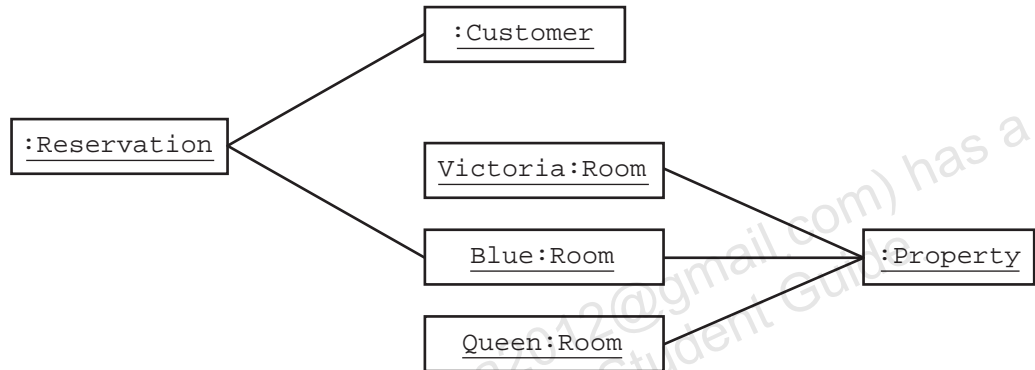
An object node might also include attributes. Object nodes can include an attributes compartment. Each attribute is listed on a separate line with an equal sign (=) to separate the name from the value. Figure 7-16 shows this notation.



**Figure 7-16** An Object Node With Attributes

## Links

In Object diagrams, each link is unique and is one-to-one with respect to the participants. A link is a unique instance of a class association. That means that each link connects only two objects together. There is no multiplicity on these links because everything is one-to-one. For example, in Figure 7-17, the `Property` object is associated with three different rooms; each association has its own association link.



**Figure 7-17** An Object Diagram With Links

## Validating the Domain Model Using Object Diagrams

To validate the Domain model:

1. Pick one or more use cases that exercise the Domain model.

Ideally, you should create Object diagrams that represent scenarios from every use case. However, this would be extremely time-consuming. Pick use cases that exercise significant portions of the Domain model.

For example, in the Hotel Reservation System, the use case “Create a Reservation” has a very broad coverage of the Domain model.

2. Pick one or more use case scenarios for the selected use cases.

Again, you should create Object diagrams that represent all scenarios for a given use case. Pick scenarios that explore a variety of situations. You want to find scenarios that push the constraints of the Domain model.

For example, in the “Create a Reservation” use case you should explore a scenario that is simple, such as reserving one room, and others that are more complex, such as reserving multiple rooms.

3. Walk through each scenario (separately), and construct the objects (with data) mentioned in the scenario.

Because scenarios are specific, they include a lot of detail. This detail is very useful. In “Creating a Scenario Object Diagram” on page 7-20, you will see how to construct an Object diagram for a use case scenario.

4. Compare each Object diagram against the Domain model to see if any association constraints are violated.

By comparing Object diagrams from a variety of use case scenarios you will see if the Domain model has been incorrectly constrained. In “Comparing Object Diagrams to Validate the Domain Model” on page 7-25, you will see how to perform this validation.

### Creating a Scenario Object Diagram

The following figures demonstrate the process of creating an Object diagram from a use case scenario. Each paragraph in the scenario represents a *time step* in the scenario. You can build the Object diagram as a sequence, one for each time step.



The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel. The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required.

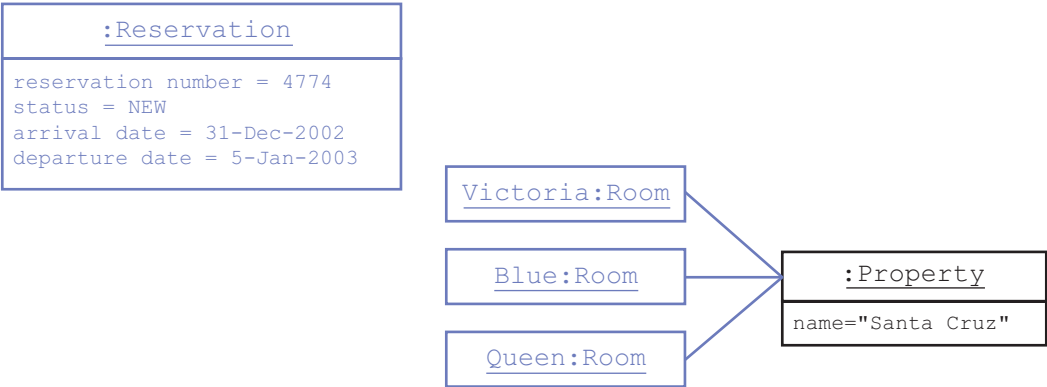
At the beginning of this scenario, the only thing known is the Property object of hotel that the customer wishes to book a room. Figure 7-18 shows this rudimentary Object diagram.



**Figure 7-18** Step 1 – Create Reservation Scenario 1

The booking agent then submits the entered details. The system finds rooms that will be available during the period of the reservation.

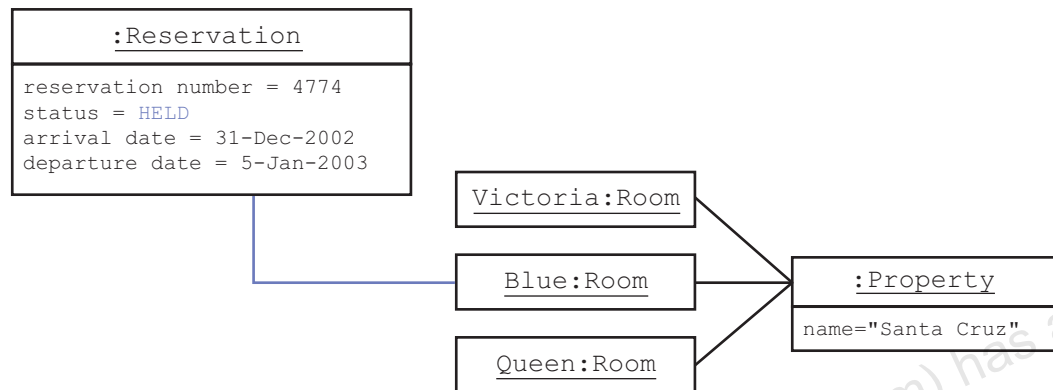
In this step of the scenario, a new Reservation object is created in the system with the arrival and departure dates filled in. Also, the booking agent has performed a search for rooms that would be available at the property. This search returned three rooms: Victoria, Blue, and Queen. The Object diagram for the scenario is expanded to include these objects. Figure 7-19 shows this. Note that this is not the same Object diagram as the previous diagram (Figure 7-18 on page 7-21) because each diagram is a snapshot in time; these represent snapshots at two different times.



**Figure 7-19** Step 2 – Create Reservation Scenario 1

The system allocates the required number and type of rooms from the available rooms. The system responds that the specified rooms are available, returns the provisional reservation number, and marks the reservation as “held”.

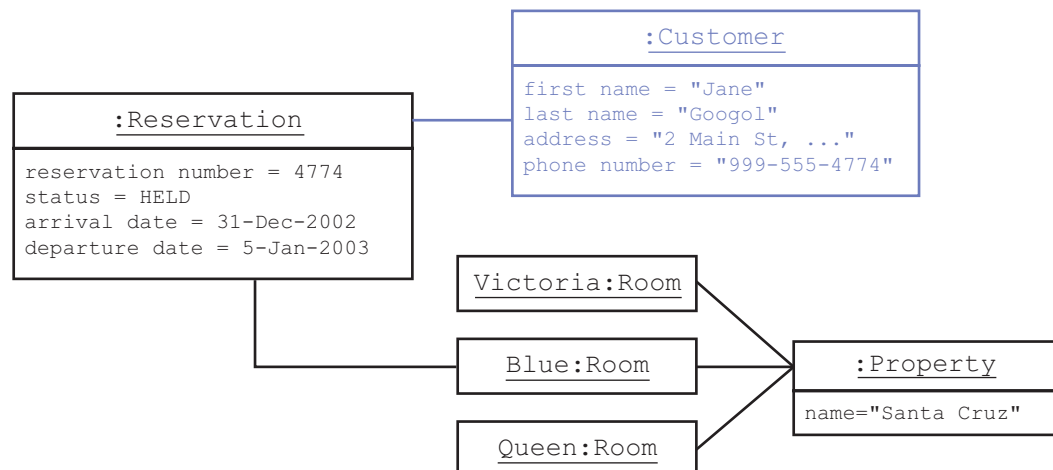
In this step, the system allocates one of the available rooms. An association is added to connect the reservation to the room. Figure 7-20 shows this Object diagram.



**Figure 7-20** Step 3 – Create Reservation Scenario 1

The booking agent accepts the room offered. The booking agent selects that the customer has visited one of the hotels in this group before, and enters the zip code and customer name. The system finds and returns a list of matching customers with full address details. The booking agent selects one of the customers as being the valid customer. The system assigns this customer to the reservation.

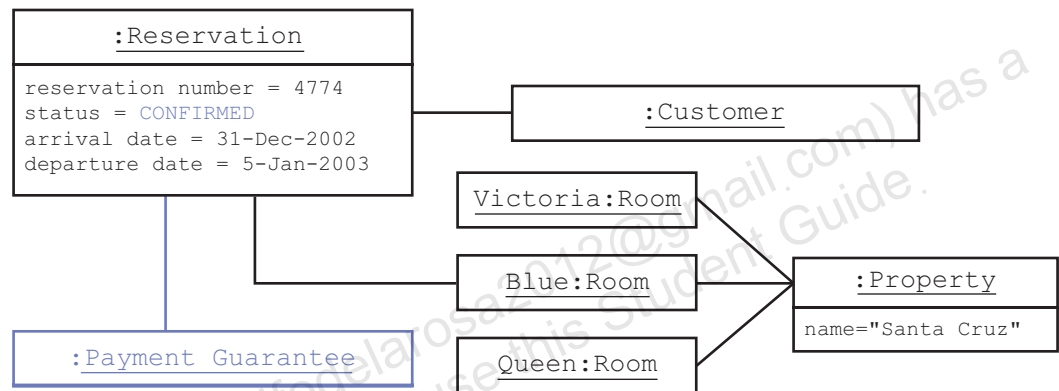
In this step, the Customer object is retrieved and associated with the reservation. Figure 7-21 shows this diagram.



**Figure 7-21** Step 4 – Create Reservation Scenario 1

The booking agent performs a payment guarantee check. This check is successful. The system assigns the payment guarantee to the reservation and changes the state of the reservation to “confirmed”. The system returns the reservation ID and booking details.

In this step, a sub-use case is invoked to ensure that the payment is guaranteed. The details of this sub-use case can be handled in separate scenarios. A Payment Guarantee object is created and associated with the reservation. The status of the reservation is changed to CONFIRMED. Figure 7-22 shows this diagram.

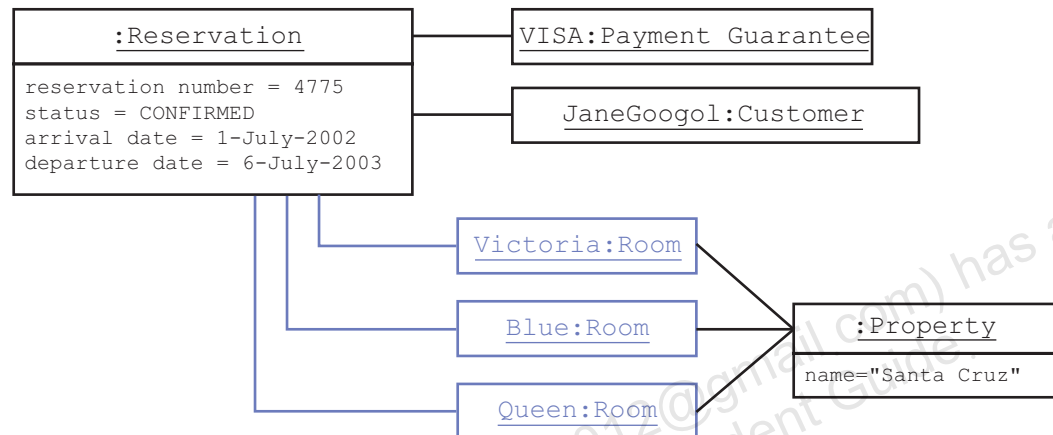


**Figure 7-22** Step 5 – Create Reservation Scenario 1

This Use Case scenario is now complete.

## Create Reservation Scenario 2

You have seen a walk through of a Use Case scenario to generate an Object diagram. Another “Create a Reservation” scenario has the Actor making a reservation for a small family reunion in which three rooms are booked. Figure 7-23 shows the Object diagram this scenario creates.



**Figure 7-23** Create Reservation Scenario 2

By comparing these two scenarios, you will see how to validate the Domain model and finds errors in the model.

## Comparing Object Diagrams to Validate the Domain Model

To validate the Domain model, compare the Class diagram with the scenario Object diagrams.

- Are there attributes or responsibilities mentioned in a scenario that are not listed in the Domain model?

Data in a scenario should be reflected in some object of the scenario Object diagram. Verify that the data attributes in the objects have corresponding data attributes in the Class diagram.

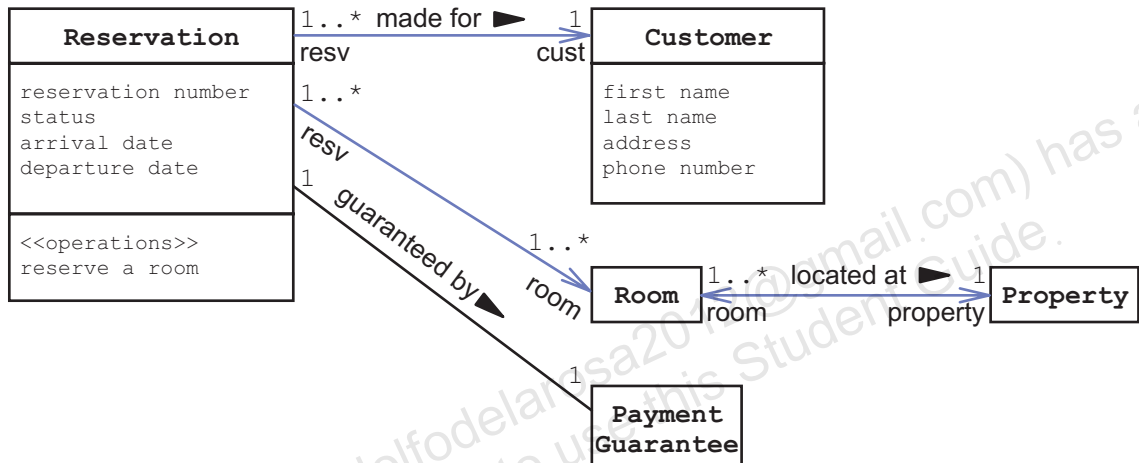
- Are there associations in the Object diagrams that do not exist in the Domain model?

Every association in the scenario Object diagrams must have a corresponding association in the Domain model. If you find an association that does not exist in the Domain model, ask the domain expert which diagram is correct, and make the necessary changes to the inaccurate diagram.

- Are there scenarios in which the multiplicity of a relationship is wrong?

For every Object diagram, count the number of associations for a given type of association, and verify that this number is valid for the multiplicity constraints specified in the Domain model.

The first scenario Object diagram verifies the Domain model, but the second scenario identifies a problem with the Domain model. The multiplicity of “Reservation reserves a Room” currently states that only one room can be reserved per reservation (see Figure 7-12 on page 7-15). The second scenario shows multiple rooms reserved for a single reservation. Therefore, the multiplicity label next to the Room class must be changed to 1..\*. Figure 7-24 shows this change.



**Figure 7-24** Revised Domain Model for the Hotel Reservation System

## Summary

In this module, you were introduced to the process of creating a Domain model. Here are a few important concepts:

- Use the Domain model to provide a static view of the key abstractions for the problem domain.
- Use the UML Class diagram to represent the Domain model.
- Validate the Domain model by creating Object diagrams from use case scenarios to see if the network of objects fits the association constraints specified by the Domain model.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.



# Transitioning from Analysis to Design Using Interaction Diagrams

---

## Objectives

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

- Explain the purpose and elements of the Design model
- Identify the essential elements of a UML Communication diagram
- Create a Communication diagram view of the Design model
- Identify the essential elements of a UML Sequence diagram
- Create a Sequence diagram view of the Design model

## Additional Resources

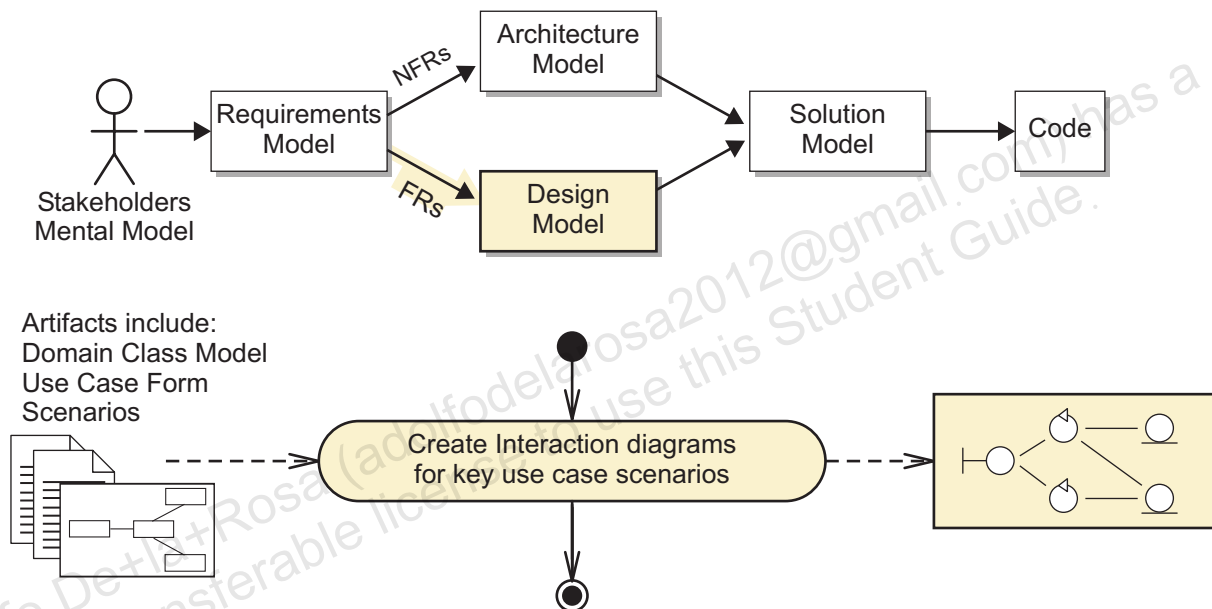


**Additional resources** – The following references provide additional information on the topics described in this module:

- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999.
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Harlow: Addison Wesley Longman, Inc., 1993.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.
- The Object Management Group. “Unified Modeling Language (UML), Version 2.2”  
[<http://www.omg.org/technology/documents/formal/uml.htm>].
- Rosenberg, Doug, Kendall Scott. *Use Case Driven Object Modeling with UML (A Practical Approach)*. Reading: Addison Wesley Longman, Inc., 1999.
- Rumbaugh, James, Jacobson Ivor, Booch Grady. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004.
- The Object Management Group. “OMG Unified Modeling Language™(OMG UML), Superstructure,”  
[<http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>], Version 2.2, February 2009.

## Process Map

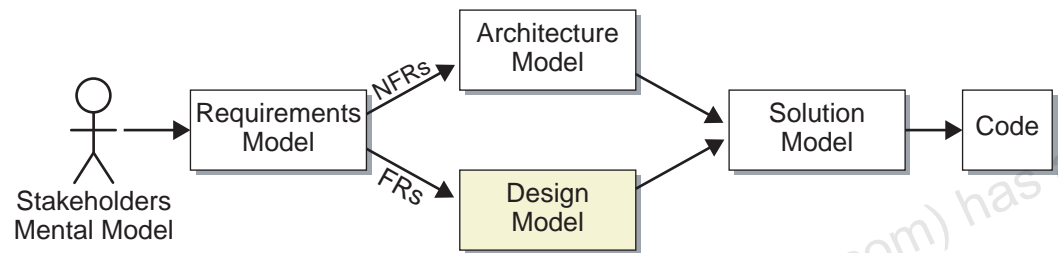
This module covers the first step in the Design workflow: creating the Design model to realize a use case. Figure 8-1 shows the activities and artifacts covered in this module. This module was placed before the Architecture workflow modules because you need to understand the purpose and structure of the Design model before creating the Architecture model.



**Figure 8-1** Interaction Diagrams Process Map

## Introducing the Design Model

A Design model is created from the functional requirements of the Requirements model. The Design model is a realization of a use case; it describes the types of components required to perform the use case. The Design model is merged with the Architecture model to produce the Solution model. Figure 8-2 illustrates this.



**Figure 8-2** Introducing the Design Model

## Interaction Diagrams

UML Interaction diagrams are the collective name for the following diagrams:

- Sequence diagrams
- Communication diagrams

Formerly known as Collaboration diagrams

- Interaction Overview diagrams

A combination of an Activity diagram and a Sequence diagram fragments

Each UML Interaction diagram is used to show the sequence of interactions that occur between objects during:

- One or two use case scenarios
- A fragment of one use case scenario

Interaction diagrams highlight the following:

- Which of the objects interact
- What messages are communicated between objects
- The sequence of the interactions

While creating an interaction diagram, you often discover the need for new classes which will assist in the control and co-ordination of the use case behavior. You may also discover new methods and attributes for the business domain classes, that were not discovered during the analysis phase.

UML Interaction diagrams might also be used to show the sequence of interactions that occur between:

- Systems
- Subsystems

Interaction diagrams are primarily used to show interaction between specific objects, however they are useful for showing interactions between composite objects, that include subsystems and systems. For example, you can use Interaction diagrams to show interactions between an ATM, the Bank that owns the ATM, and the card users Bank.

## Comparing Analysis and Design

Analysis helps you model *what* is known about a business process that the system must support:

- Use cases  
The behavior of the system relative to the user
- Domain model  
The entities involved in the business processes

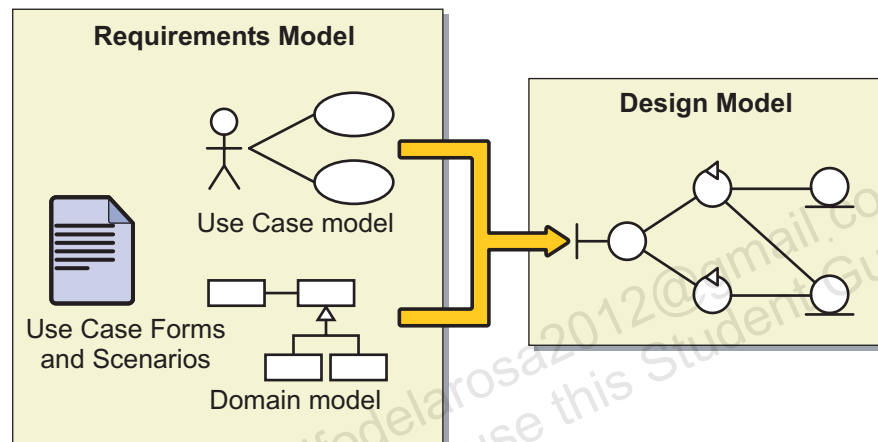
Design helps you model *how* the system will support the business processes. The Design model consists of:

- Boundary (UI) components  
These components interact directly with the user. These components affect the system by accessing functions of the Service and Entity components.
- Service components  
These components perform business operations and use case workflow management.
- Entity components  
These components correspond to the entities in the Domain model.

## Robustness Analysis

The components identified during robustness analysis are called design components.

Robustness analysis is a process that assists in identifying design components that would be required in the Design model. Figure 8-3 illustrates this.



**Figure 8-3** The Design Model Is Derived From the Requirements Model

Robustness analysis is the name given to the process that creates the Design model. This name is not widely used in the industry, but the process has deep roots starting with Jacobson's OOSE methodology.

Robustness analysis uses these inputs:

- A use case  
A use case is selected to be designed. You could perform Robustness analysis on multiple use cases at once, but that would lead to a rather large Design model. It is better to create multiple, small Design models; one for each use case of the system.
- The use case scenarios for that use case  
Robustness analysis is like CRC analysis in that you will perform a *walk through* of use case scenarios to construct the Design model.
- The use case Activity diagram (if available) for that use case  
You can use an Activity diagram for a use case instead of walking through each scenario.
- The Domain model

The Domain model is the source of entity components for the Design model. Not all Domain entities are used in every use case, therefore, not all of the Domain entities will appear in any given Design model. Only the entities that the use case being designed uses will appear in the Design model.

The Design model is usually captured in UML Interaction diagrams with design components such as Boundary, Service, and Entity components.

## Boundary Components

“A boundary class (component) is used to model interaction between the system and its actors (that is, users and external systems).” (Jacobson, Booch, and Rumbaugh page 183)

A Boundary component is represented as a circle with a T-shaped line sticking to the side facing the actor. Figure 8-4 shows this.



**Figure 8-4** An Example Boundary Component

Characteristics of Boundary components include:

- Abstractions of UI screens, sensors, communication interfaces, and so on.  
Boundary components exist to interact with the outside world whether that is a human actor or a machine actor. These components can represent any contact to the outside of the system.
- High-level UI components.  
Boundary components must remain at a high-level. Do not decompose the UI into individual GUI widgets, such as text fields, labels, sliders, radio buttons, and so on.
- Every boundary component must be associated with at least one actor.

Boundary components exist to interact with actors. A Boundary component must be associated with at least one actor, but it can also interact with multiple actors. If a Boundary component is designed, yet is not associated with any actor, then that component serves no purpose.

## Service Components

“Control (Service) classes (components) represent coordination, sequencing, transactions, and control of other objects and are often used to encapsulate control related to a specific use case.” (Jacobson, Booch, and Rumbaugh page 185)

A Service component is represented as circle with an arrow on the edge of the circle. This is shown in Figure 8-5.



**Figure 8-5** An Example Service Component

Jacobson called these components Control classes, but the term *control* can mean several different things. This course uses the term *service* to refer to these components as they relate back to the four fundamental application components discussed in Figure 12-2 “Generic Application Components” on page 12-6. The term *control* or *controller* usually refers to components that handle user actions.

Characteristics of Service components include:

- Coordinate control flow  
Service components have many purposes. However, at the top-level these components usually correspond to use cases of the system. That is to say that Service components manage use case workflows. Service components can also implement common functionality between multiple use cases. Service components can also implement complex algorithms.
- Isolate any changes in workflow from the boundary and entity components

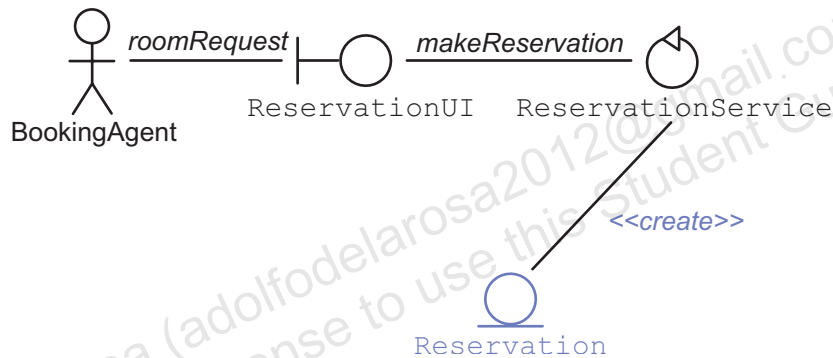


You should separate UI behaviors from workflow behaviors, because the system will be more maintainable if these different behaviors are encapsulated in different components.

## Entity Components

“An entity class (component) is used to model information that is long-lived and often persistent.” (Jacobson, Booch, and Rumbaugh page 184)

An Entity component is represented as circle with line at the bottom of the circle. Figure 8-6 shows this.



**Figure 8-6** An Example Entity Component

Characteristics of Entity components are:

- Entities usually correspond to domain objects.  
There is usually a one-to-one mapping between the Domain model and the Entity components. However, auxiliary entities might be discovered during Robustness analysis.
- Most entities are persistent.  
Entities tend to be persistent objects. This is not a fixed rule; some entities might be created to hold results from other entities.
- Entities can have complex behavior.

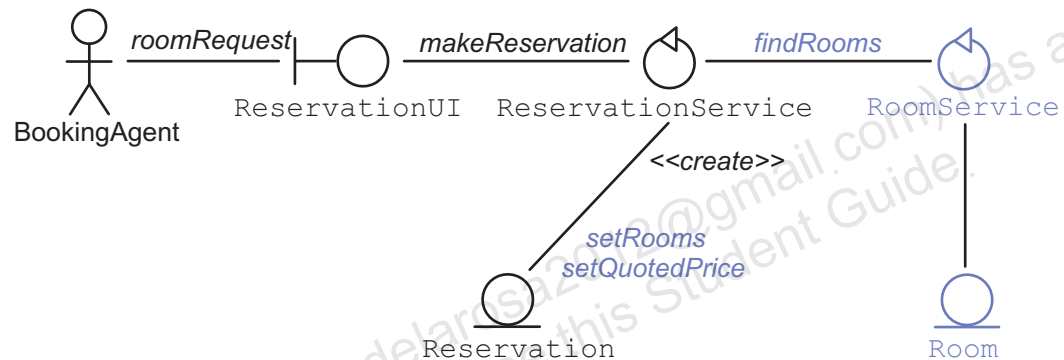
From this description you might think that entities only exist to hold attributes. However, entities can have complex behavior. There is a trade-off between putting this behavior in a Service component or in an Entity component. A general guideline is that if the behavior involves the interaction of multiple entities, then it should probably be in a Service component.

## Service and Entity Components

An Entity component will often have a corresponding Service component.

- Each service object will often control its corresponding entity object.
- One service object can delegate control to another more coherent service object.

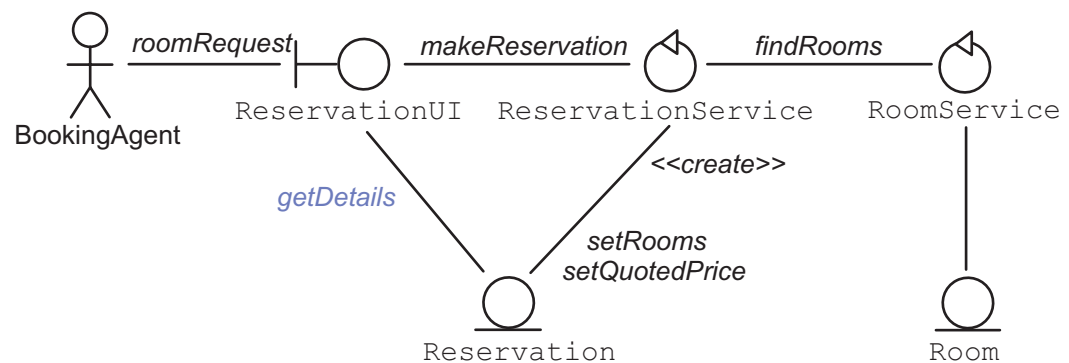
Figure 8-7 shows that the ReservationService object delegates the task of finding rooms to the RoomService object, but manages the Reservation object directly.



**Figure 8-7** An example of coherent services

## Boundary And Entity Components

A Boundary component can often retrieve the attributes of an Entity component. Figure 8-8 shows an example of a Boundary object accessing the details from the Reservation entity object.



**Figure 8-8** An example of a Boundary object accessing data from an Entity object

## Describing the Robustness Analysis Process

The Robustness analysis process is as follows:

1. Select a use case.

Robustness analysis is driven by a use case. Select a single use case to develop into a Design model.

2. Construct a Communication diagram or Sequence diagram that satisfies the activities of the use case.

Using the use case Activity diagram (or scenarios), analyze each action in the Activity diagram, and perform the following steps:

- a. Identify design components that support the activities of the use case.

If the action refers to user interaction, create a Boundary component to handle this interaction. If the action refers to some action taken by the system, use a Service component. If the action refers to a responsibility of a Domain object, use an Entity component.

- b. Draw the associations between these components.

Identify the associations between components to perform the action described in the Activity diagram.

- c. Label the associations with messages.

Create labels on the associations to describe the messages sent between components for performing the action.

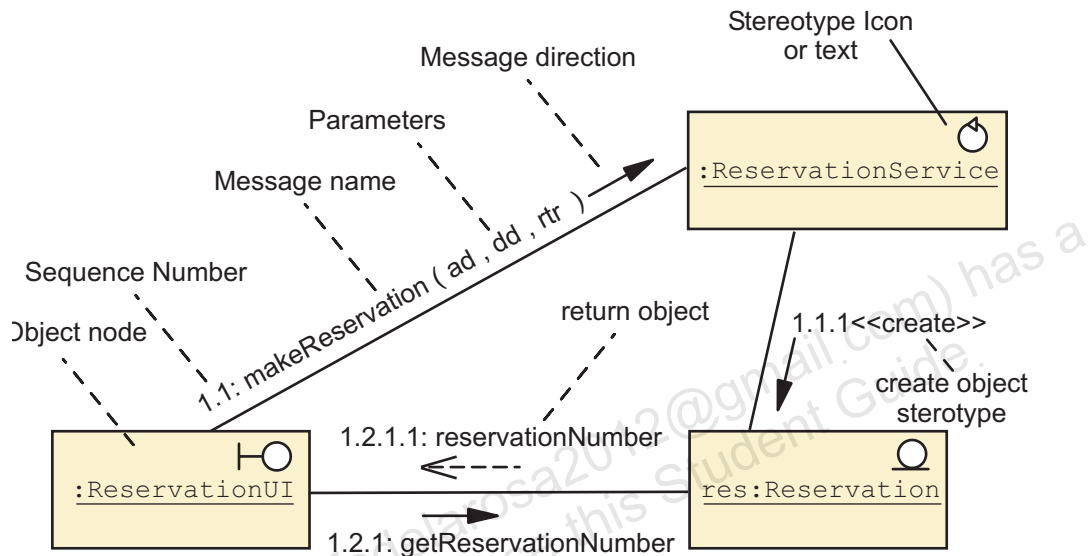
3. Convert the Communication diagram into a Sequence diagram, or vice versa, for an alternate view (optional).

This step enables you to transform the Communication diagram into a Sequence diagram, which provides a time-ordered view of the messages between the collaborating objects.

This is a very high-level description of the Robustness analysis process. The following sections in this module describe this process in more detail. The next section describes the features of the UML Communication diagram.

## Identifying the Elements of a Communication Diagram

A *Communication diagram* represents the objects of a system, their relationships, and the messages sent between objects. Figure 8-9 shows a Communication diagram.



**Figure 8-9** An Example of a Communication Diagram

Message arrows show the direction in which the objects collaborate. For example, the arrow labeled “1.1:makeReservation” indicates that the ReservationUI object sends the makeReservation message to the ReservationService object.

A Message arrow can indicate:

- A method call – Signified by a solid line and filled arrowhead
- A return from a synchronous method call – Signified by a dashed line and an open arrowhead
- An asynchronous messages – Signified by a solid line and an open arrowhead

A Sequence label indicates the following:

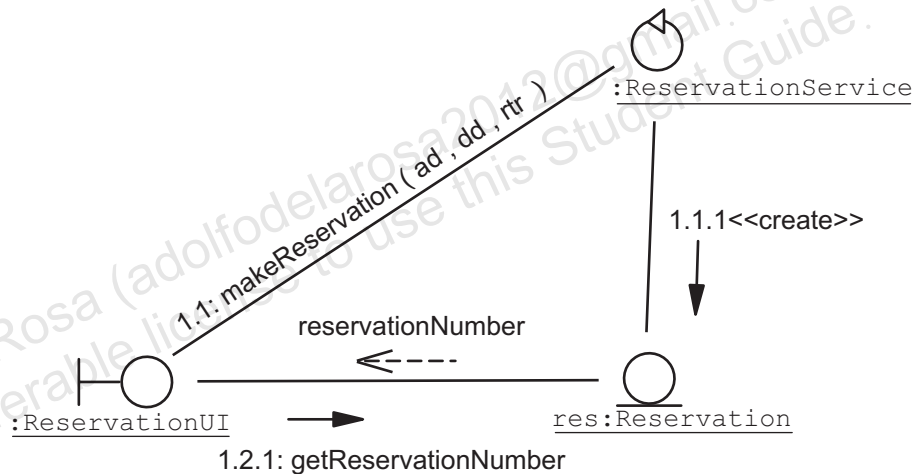
- The order of the message

The first portion of a label indicates a hierarchical order in which the message is issued in the communication. For example, in the label “1.1.1:«create»”, “1.1.1” indicates the order in which this

message is sent. The «create» message is sent after the makeReservation message, but before the getReservationNumber message.

- The activity the message will invoke  
The second portion of a label indicates the actual message that is being invoked on the receiving object. For example, in the label “1.1:makeReservation(ad,dd,rtr)”, “makeReservation” indicates that the makeReservation method is invoked on the ReservationService object.

In Robustness analysis, you will see a variation in the Communication diagrams such that the object node stereotypes are used as icons for the node. Figure 8-10 shows an example; this diagram is similar to the diagram in Figure 8-9 on page 8-12.



**Figure 8-10** A Variation Using Stereotype Icons

## Creating a Communication Diagram

Robustness analysis can be performed by completing the following steps on a selected use case.

1. Place the actor in the Communication diagram.
2. Analyze the Use Case form or the Activity diagram for the use case. For every action in the use case:
  - a. Identify and add a Boundary component.
  - b. Identify and add a Service component.
  - c. Identify and add an Entity component.
  - d. Identify and add further Methods in the Service and Entity components to satisfy the interactions in the use case.

The following sections describe the steps to create a Communication diagram for the “Create a Reservation” use case.

### Step 1– Place the Actor in the Diagram

Every use case starts with the actor that uses the use case. For example, in the “Create Reservation” use case the BookingAgent is one of the actors for this use case. Figure 8-11 shows that you should place the BookingAgent actor icon at the far left of the Communication diagram.



**Figure 8-11** Step 1 – Place the Actor in the Communication Diagram

## Step 2a – Identify Boundary Components

By analyzing an action in the Flow of Events, you must identify the three types of design components. Boundary components are fairly easy to identify. These components enable the actor to interact with the system. Actions that have the actor do something with the system requires one or more Boundary components. For example, the “BookingAgent enters the arrival date, departure date, requested types of rooms, and then submits these to the boundary in ‘roomRequest’ message. Figure 8-12 shows this.



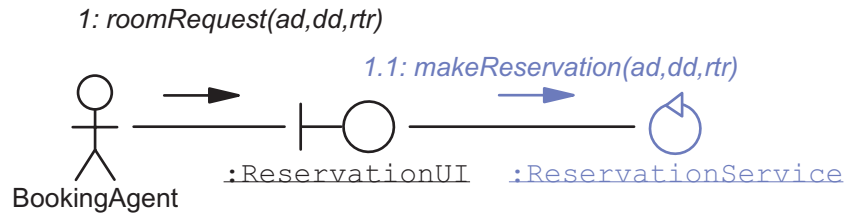
**Figure 8-12** Step 2a – BookingAgent Interacts With the ReservationUI Component

This diagram shows an association between the actor and the ReservationUI component. The message arrow and label over the association indicates the action and its sequence within the Flow of Events. Note that these messages are not specified by the use case Flow of Events. You must identify how the system satisfies the actions in the use case. This requires imagination and a deep understanding of the purpose of the use case. A User Interface prototype can also be helpful during Robustness analysis because the Boundary components are already defined by the screens of the UI prototype.

Sanity checking of the data is usually performed in the boundary component. For example the dates will be checked for invalid days, months, and years.

## Step 2b – Identify Service Components

The ReservationUI components need to delegate the request to business methods which are modeled with a Service component. For example, the roomRequest method in ReservationUI calls the makeReservation method in the ReservationService component, as Figure 8-13 shows.



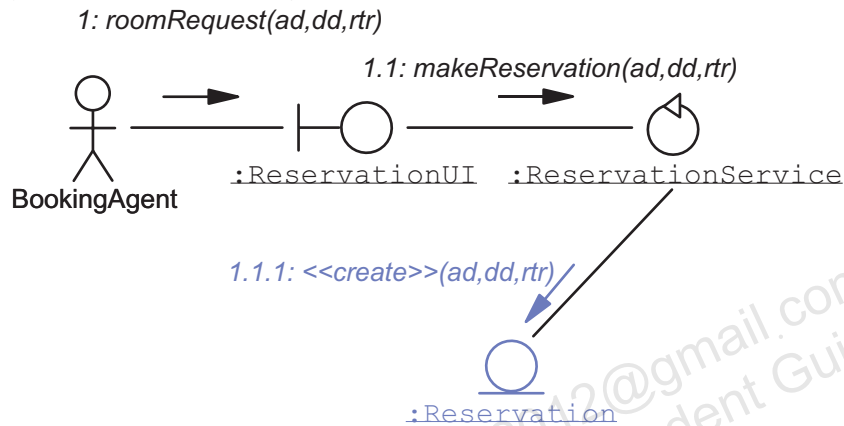
**Figure 8-13** Step 2b – The ReservationUI Component delegates some of its behavior to a business method in the ReservationService Component

This ReservationService component is fairly generic. It will support most operations for the “Create a Reservation” use case. However, you can also have ReservationUI component interact with other services as necessary.



## Step 2c – Identify Entity Components

Eventually, the system needs to create and manipulate entities. These entities are modeled with Entity components. For example, the ReservationService component will create a Reservation Entity component and passes the arrival date, departure date, and requested type of rooms, as Figure 8-14 shows.



**Figure 8-14** Step 2c – The ReservationService Component creates the Reservation Entity Component

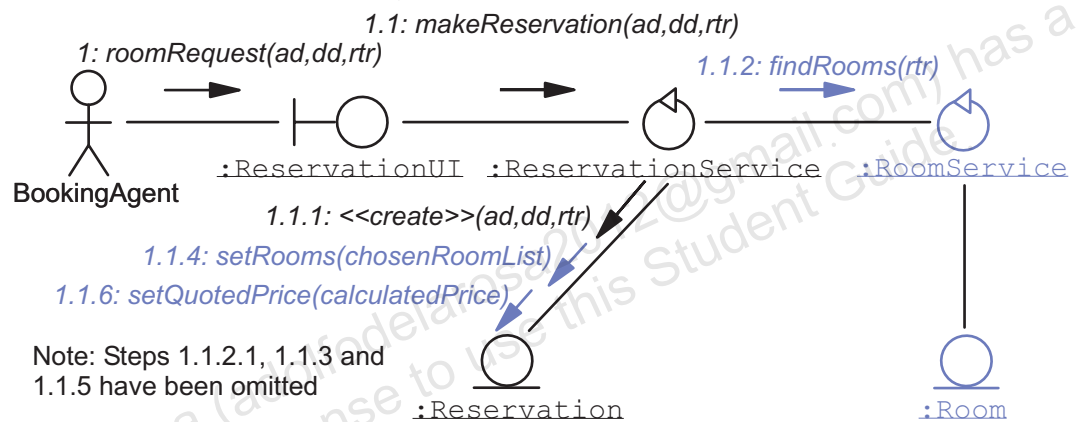


**Note** – At this conceptual stage, the Design model does not effectively represent how the system will interact with a database. For now, the service components will appear to perform all database operations.

## Step 2d – Identify Additional Interactions

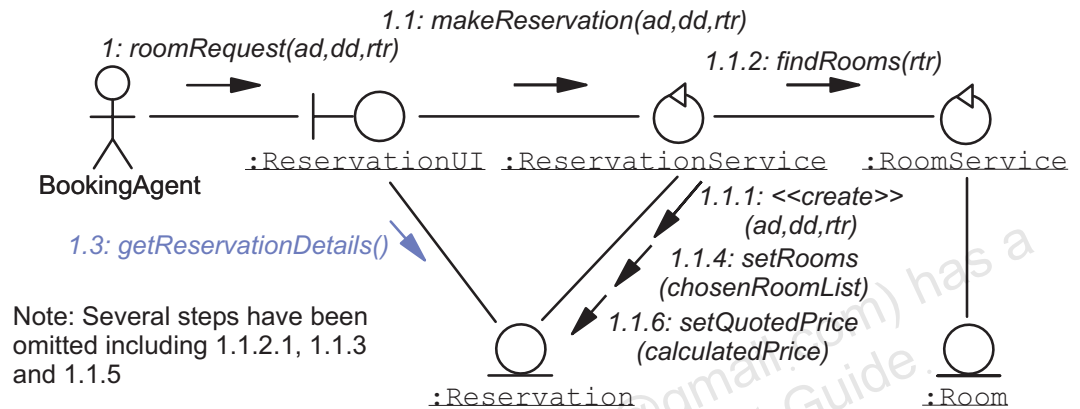
The previous pages demonstrated Robustness analysis for a single action in the Flow of Events for the use case. Continue this process for every action in the Flow of Events. This diagram will elaborate the Design model to completely handle the use case.

The makeReservation method uses the findRoom method to find rooms of the required type. After finding and choosing free rooms (not shown in Figure 8-15), the setRooms method assigns the rooms to the Reservation entity. The calculated price (not shown in Figure 8-15) is assigned to the Reservation as shown in Figure 8-15



**Figure 8-15** Additional Interactions between Services and Entity Components

The makeReservation method of the ReservationService object returns the Reservation object (not shown in Figure 8-16) to the ReservationUI. The roomRequest method of ReservationUI then calls the getReservationDetails method of the Reservation object, which returns the details of the reservation. These will then be notified to the booking agent (not shown in Figure 8-16), as shown in Figure 8-16.

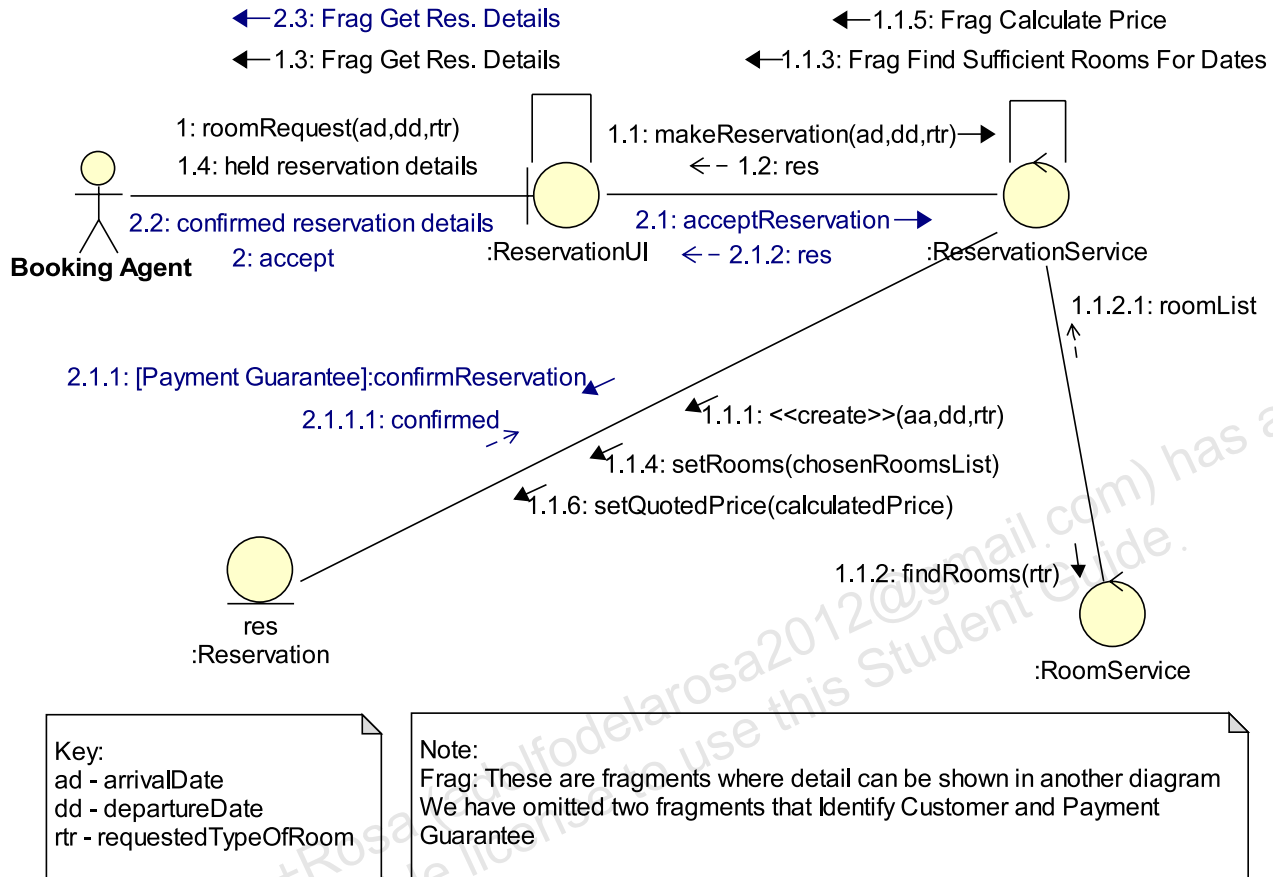


**Figure 8-16** Additional Interactions between Boundary and Entity Components

## Communication Diagram Examples

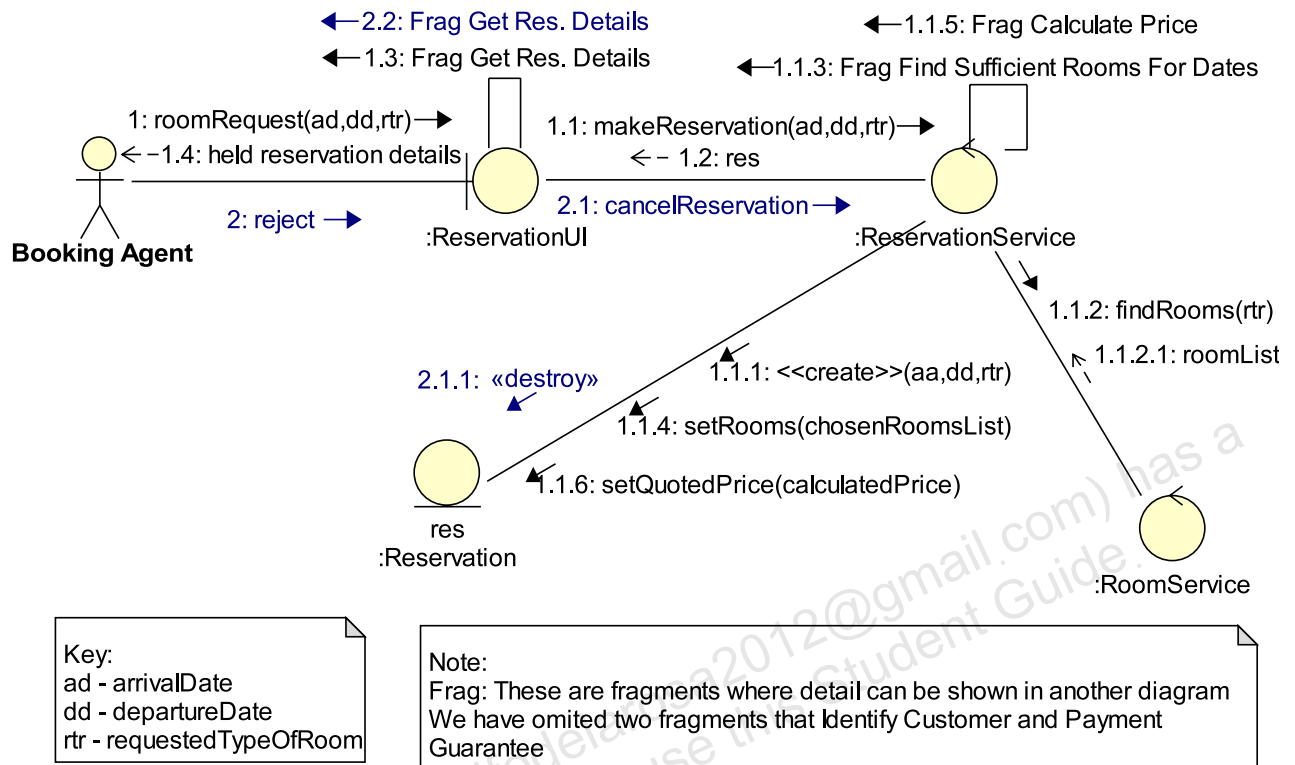
The following two Communication diagrams show a more detailed view of the CreateReservation:

- Figure 8-17 shows a Primary (successful) scenario.



**Figure 8-17** Example of a Primary (successful) Scenario Communication Diagram

- Figure 8-18 shows a Secondary (unsuccessful) scenario, where Rooms offered are rejected by the booking agent:



**Figure 8-18** An Example of a Secondary (unsuccessful) Scenario Communication Diagram

## Sequence Diagrams

Sequence diagrams are UML diagrams that:

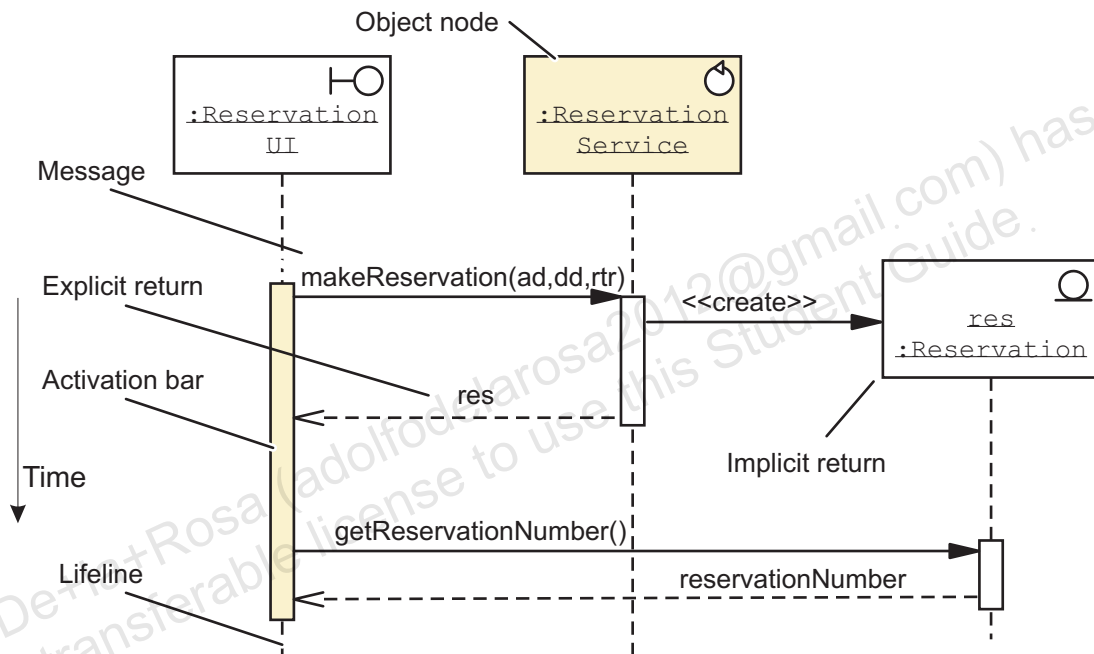
- Provide a different perspective on the interaction between objects
- Can be used instead of Communication diagrams
- Can be converted to or from a Communication diagram
- Prove to be more useful for developers
- Highlight the time ordering of the interactions

The next section describes UML Sequence diagrams.

## Identifying the Elements of a Sequence Diagram

A Sequence diagram is “A diagram that shows object interactions arranged in time sequence.” (UML v1.4, page B-17)

A *Sequence diagram* represents the objects of a system and the messages sent between objects organized in a time-ordered fashion. Figure 8-19 shows an example Sequence diagram.



**Figure 8-19** An Example Sequence Diagram

The collaborating objects are organized along the top of the diagram (creating columns). Time is organized from the top to the bottom of the diagram.

Under each object is a dashed line that indicates the duration or the *life* of the object. If the object exists throughout the duration of the activity represented by the Sequence diagram, then the lifeline should extend through the bottom of the diagram.

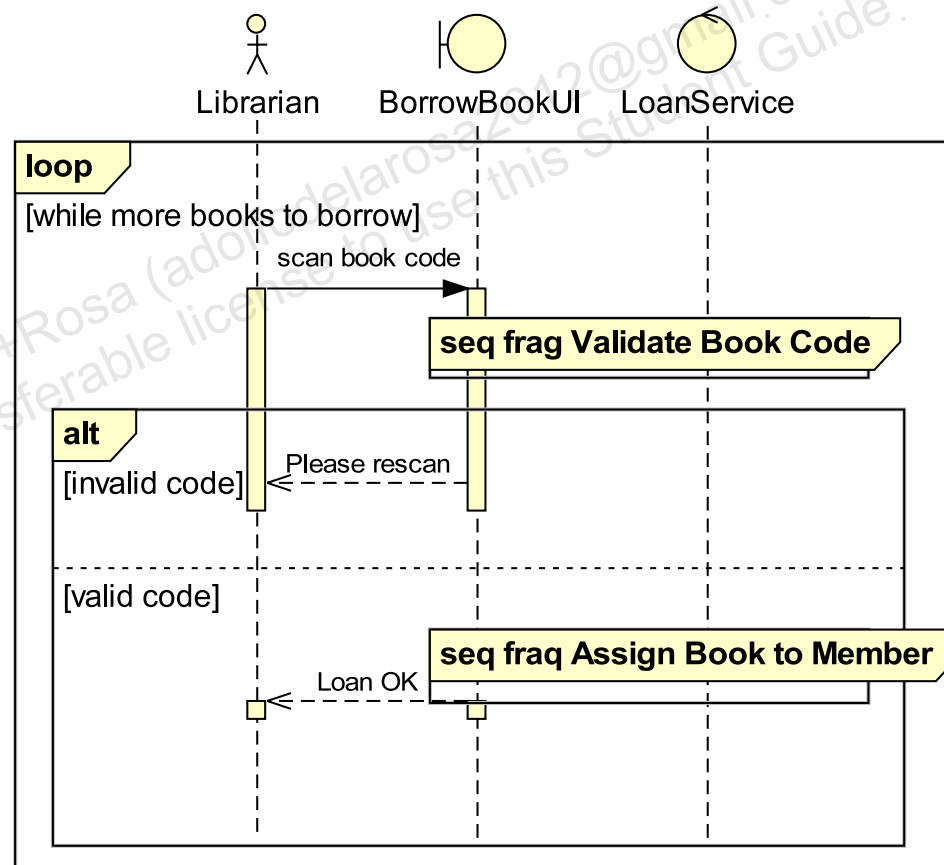
The activation bar indicates that the object is engaged in some activity. This usually occurs when a message is sent to the object. For example, the `makeReservation` message is sent to the `ReservationService` object. An activation bar starts at the point where the message arrow arrives at the object and extends down (in time) to the point when the `ReservationService` object has completed processing that message. A dashed line with a stick arrow provides an explicit indication that a message returns a value. Alternatively, you can leave out the return arrow; in this case the end of the activation bar indicates that the message processing is complete.

# Fragments

Sequence diagrams support a Fragment notation. The uses of fragments include:

- Showing sequence loops
- Showing alternative paths
- Allowing two or more scenarios to be shown on one diagram
- Showing a reference to another detailed Sequence diagram fragment
- Allowing you to break up a large diagram into several smaller diagrams

Figure 8-20 shows examples of the Fragment notation.



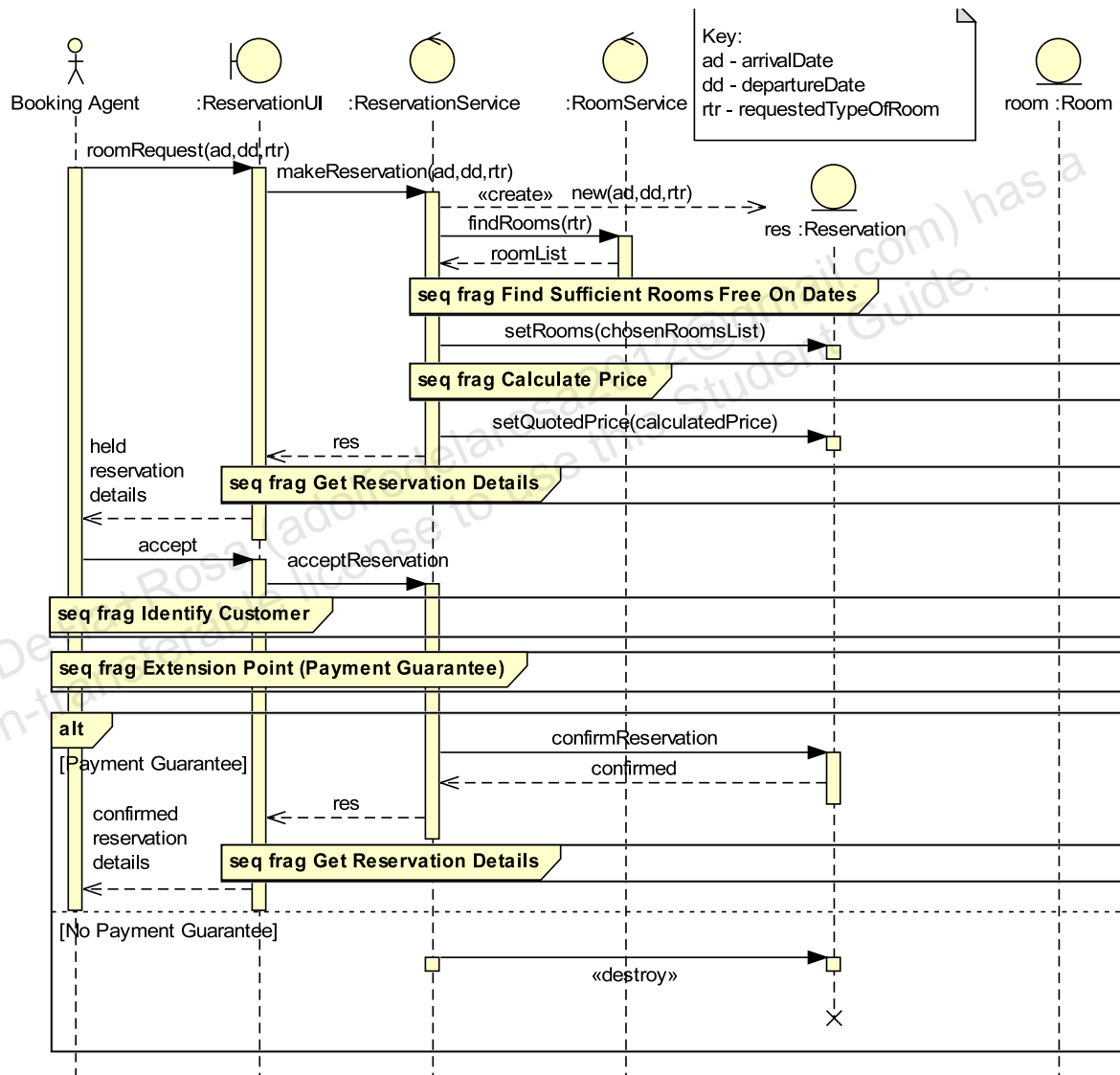
**Figure 8-20** An Example Showing a Loop, an Alt, and a Reference Fragment



# Sequence Diagram Examples

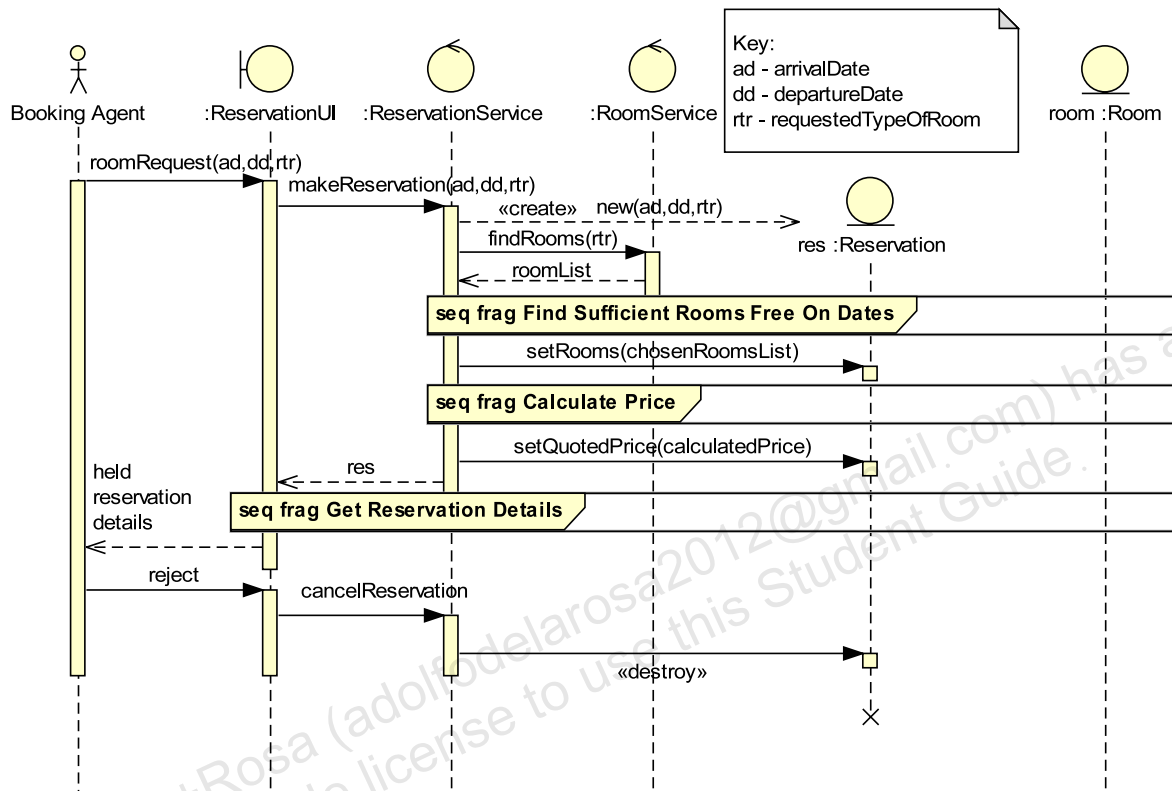
The following three Sequence diagrams show a more detailed view of the CreateReservation:

- Figure 8-21 shows the primary (successful) scenario and an secondary (unsuccessful) scenario in one diagram using an *alt* fragment.



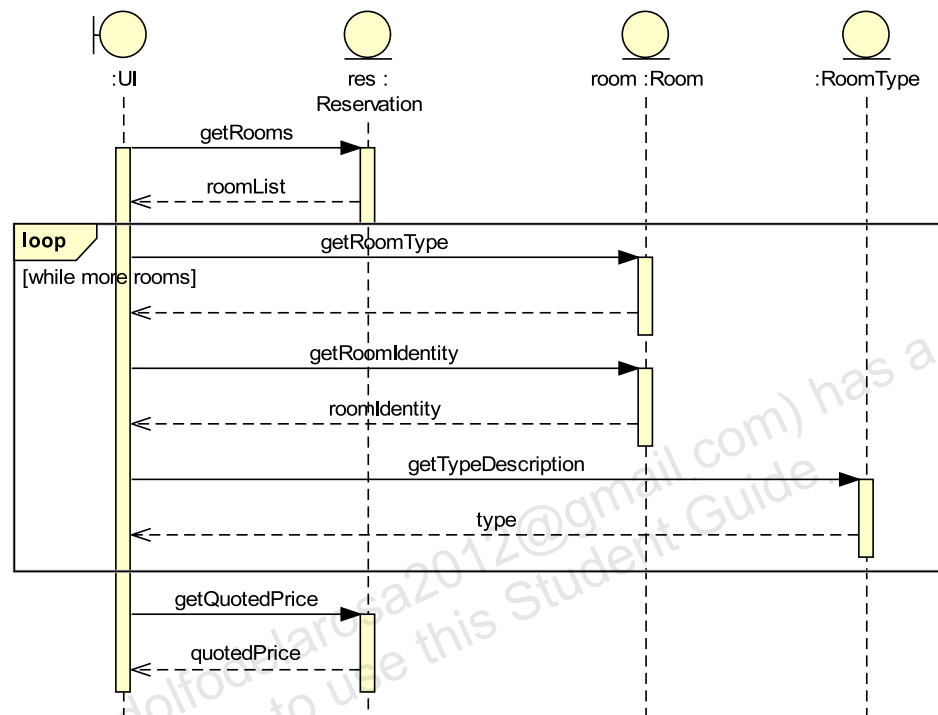
**Figure 8-21** Example of a Primary (successful) and Secondary (unsuccessful) scenarios on the same diagram

- Figure 8-22 shows a secondary (unsuccessful) scenario, where the rooms offered are rejected by the booking agent.



**Figure 8-22** Example of a Secondary Scenario Sequence Diagram

3. Figure 8-23 shows a Fragment Sequence diagram that is referenced from both Figure 8-21 and Figure 8-22. It shows the finer details of the GetReservationDetails fragment and includes a *loop* fragment.



**Figure 8-23** Example of a Sequence Diagram Fragment

## Summary

In this module, you were introduced to Robustness analysis and the Design model. The following lists a few important concepts that were covered:

- Interaction diagrams are used to identify design components that satisfy a use case.
- Object interactions can be visualized with a UML:
  - Communication diagram
  - Sequence diagram.

# Modeling Object State Using State Machine Diagrams

---

## Objectives

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

- Model object state
- Describe the essential elements of a UML State Machine diagram

## Additional Resources



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

- Folwer, Martin, with Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Rumbaugh, James, Jacobson Ivor, and Booch Grady. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004.
- The Object Management Group. “OMG Unified Modeling Language™ (OMG UML), Superstructure,”  
[<http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>],  
Version 2.2, February 2009.



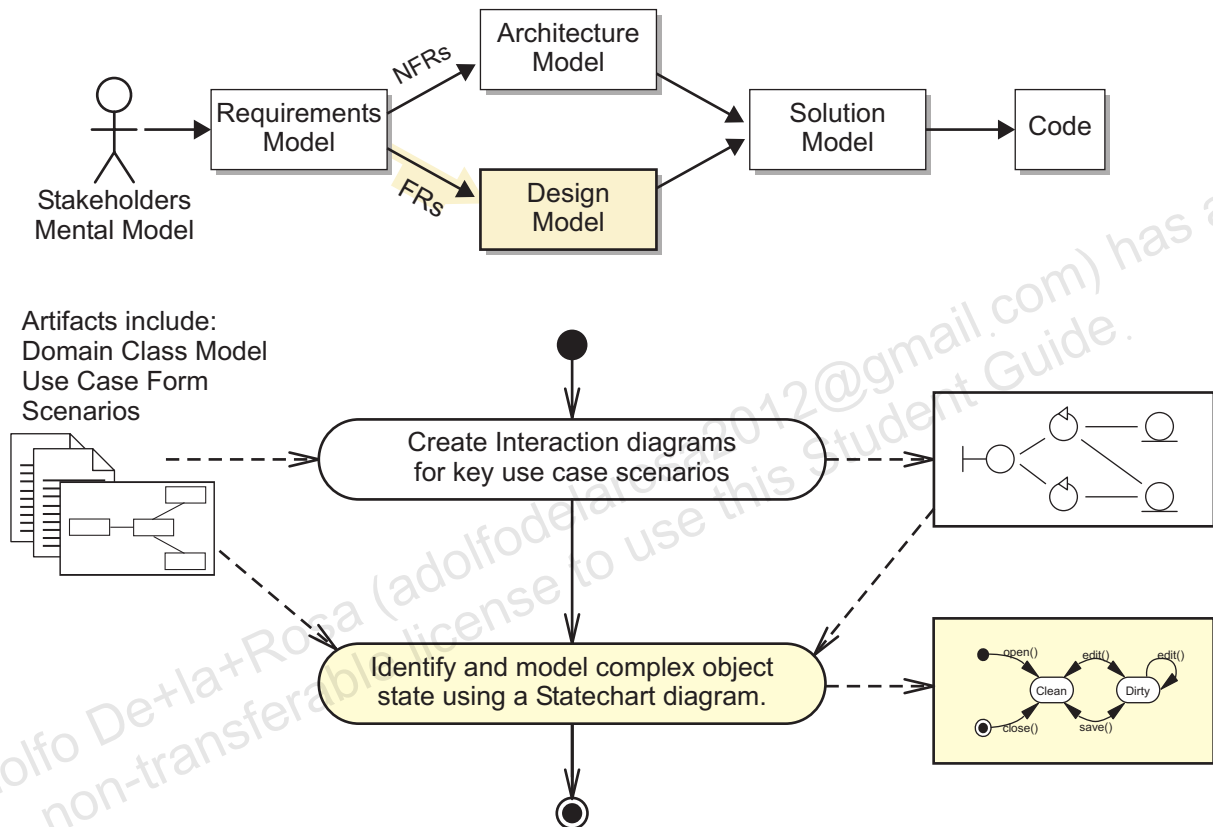
---

**Note** – The mathematical basis of the Statechart diagram is founded in the state machine theories of Meale and Moore. (See Booch, Rumbaugh, and Jacobson UML User Guide page 336).

---

# Process Map

This module describes the next step in the Design workflow: modeling the state behavior of a complex object. Figure 9-1 shows the activity and artifact discussed in this module.



**Figure 9-1** Design Workflow Process Map

# Modeling Object State

This section discusses what object state is and how you can model state information using the UML.

## Introducing Object State

State is “1a: mode or condition of being” (Webster)

There are two ways to think about object state:

- The state of an object is specific collection of attribute values for the object.

Typically people think of object state as the state of the attributes in an object. This view is certainly valid, but it misses an important understanding about object behavior.

For example, imagine a software system that is controlling a heating, ventilating, and air-conditioning system (HVAC). You could think of the state of the HVAC system in terms of whether the power is on, and whether the current (real) temperature is within a certain range (upper bound and lower bound temperatures).

- The state of an object describes the behavior of the object relative to external stimuli.

This view is behavior focused rather than data focused. In the HVAC example, there are four basic states: Initial (meaning the unit has not been turned on), Idle (the unit is on but not active), Cooling (the unit is on and cooling), and Heating (the unit is on and heating). These states talk about behaviors of the HVAC.

There are also triggers to change states. For example, if the HVAC unit is on and the real temperature is recorded as being above the upper bound, then the system should begin cooling (transition to the Cooling state).

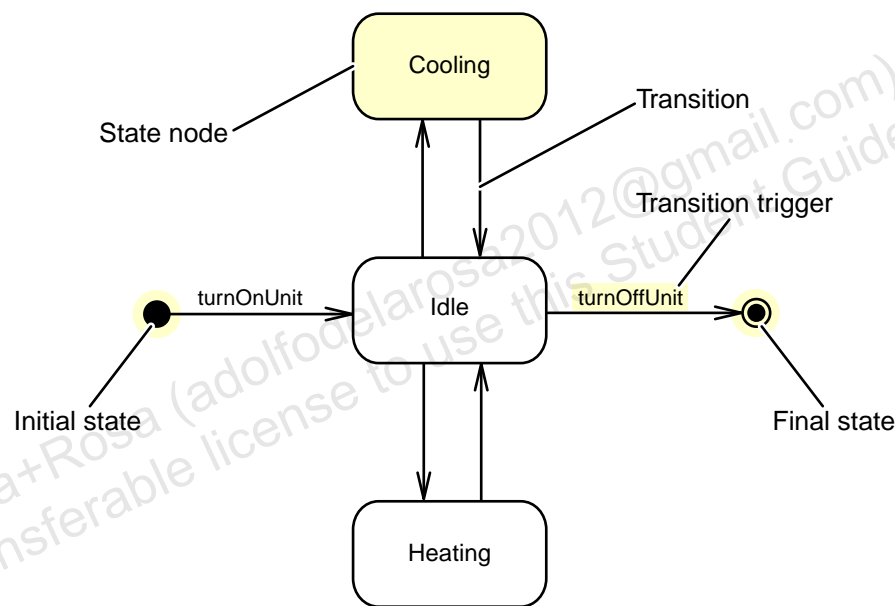
This module describes how to model and implement complex objects with state. This module will use the HVAC system example throughout.



## Identifying the Elements of a State Machine Diagram

A state machine is “A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.” (UML v1.4 page B-18)

A State Machine diagram *State Machine diagram* represents the set of states for an object or class, the activities of a given state, and the triggers that transition an object from one state to another. Figure 9-2 shows an example State Machine diagram.



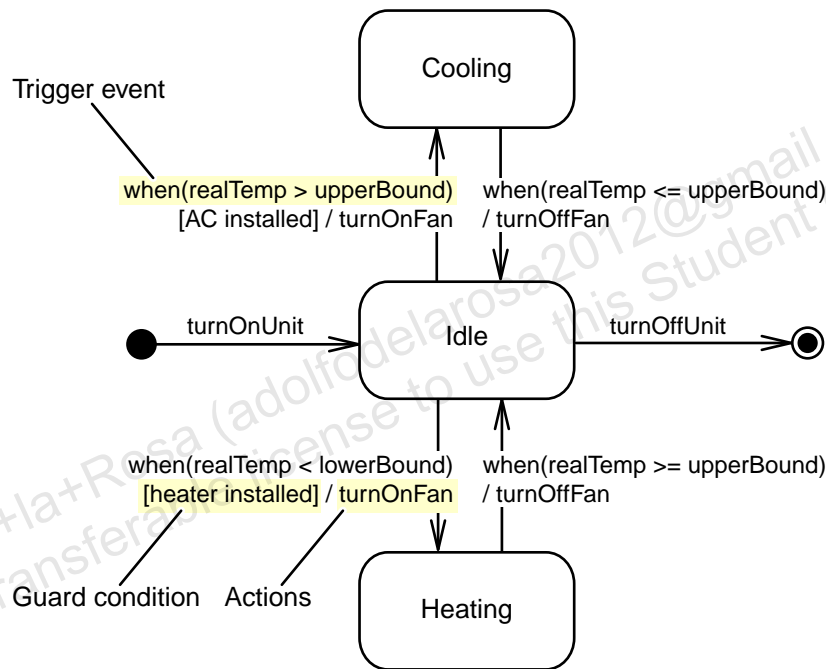
**Figure 9-2** Example State Machine Diagram

Every State Machine diagram must include Initial and Final state nodes. Typically, these nodes do not have any behavior associated with them. A state node is represented with a rounded-corner rectangle. A transition from one state to another is represented with a solid line with a stick arrowhead. A transition usually has a trigger associated with it. This trigger is usually an external stimulus on the object. In the HVAC example, turning the HVAC unit on triggers a transition from the Initial state to the Idle state.

This diagram is incomplete without the triggers on the Idle to Cooling or Heating states.

## State Transitions

A state transition represents a change of state at runtime. A state transition usually includes a trigger event declaring when an object changes from one state to another. For example, if the HVAC system is in the Idle state and the real temperature rises above the upper bound, then the HVAC object transitions to the Cooling state. A transition can also have a guard condition and a set of actions. In this example, the HVAC can only begin cooling if the AC subsystem has been installed for this particular HVAC system; if there is no AC, then no cooling can occur. There is also an action on this transition to turn the HVAC fans on. Figure 9-3 shows the state transitions for the HVAC system.



**Figure 9-3** HVAC State Machine Diagram With Transitions

## Internal Structure of State Nodes

State nodes represent a state of a single object at runtime. The internal structure of a state node can include specific actions to be taken in the case of object events that do not transition out of the given state. Figure 9-4 shows the internal structure of state nodes.



**Figure 9-4** Internal Structure of State Nodes

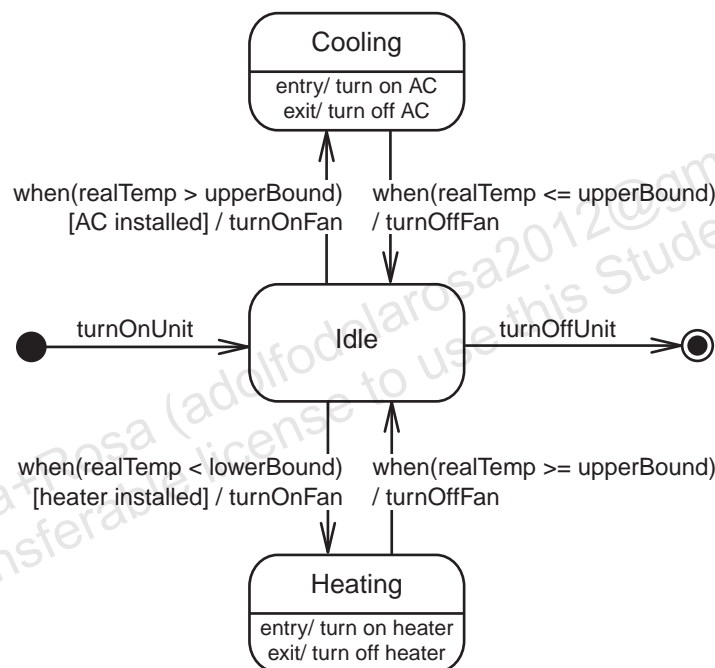
The types of events include:

- Entry – Specifies actions upon entry into the state.
- Exit – Specifies actions upon exit from the state.
- Do – Specifies ongoing actions.

You can also specify specific events with corresponding actions.

Entry, exit, and do events are specified by the UML.

Figure 9-5 shows a complete State Machine diagram for the HVAC example.



**Figure 9-5** Complete HVAC State Machine Diagram

## Creating a State Machine Diagram for a Complex Object

Creating a State Machine diagram requires a deep understanding of the object that is modeled. This analysis is beyond the scope of this course.

You can use the following steps to simplify the creation of a State Machine diagram after the states of the object are understood:

1. Draw the initial and final state for the object.
2. Draw the stable states of the object.
3. Specify the partial ordering of stable states over the lifetime of the object.
4. Specify the events that trigger the transitions between the states of the object. Specify transition actions (if any).
5. Specify the actions within a state (if any).

### Step 1 – Start With the Initial and Final States

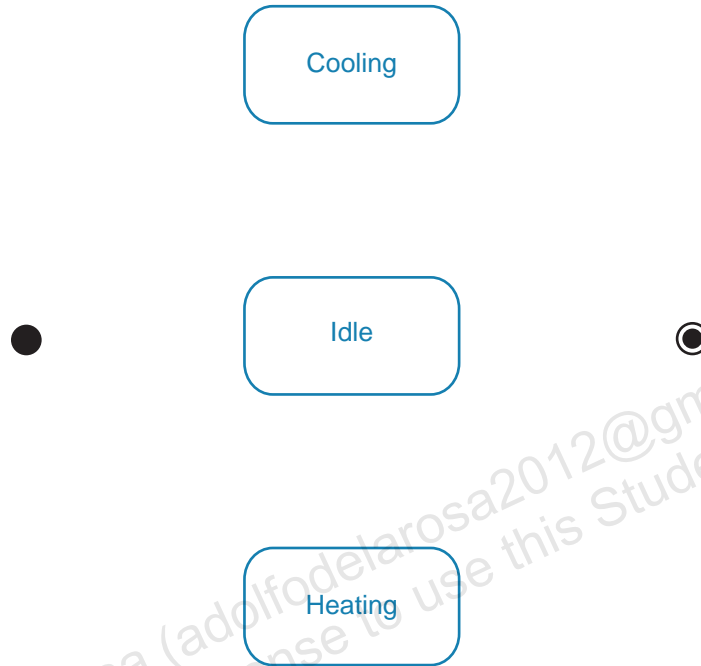
The first step is easy. Place the initial and the final state nodes in the diagram. Do not forget to place these states, because every State Machine diagram must include them. Figure 9-6 illustrates how to place the initial and final nodes.



**Figure 9-6** Step 1 – Start With the Initial and Final States

## Step 2 – Determine Stable Object States

Then determine the stable states of the object (besides the Initial and Final) and create state nodes for each of these object states. Figure 9-7 illustrates this step.

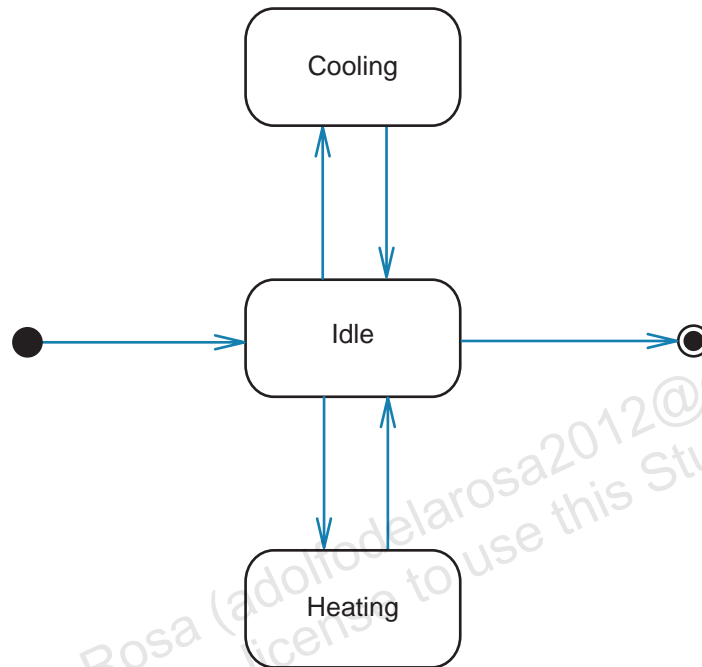


**Figure 9-7** Step 2 – Determine Stable Object States

A stable state is the set of conditions and behaviors in which an object can reside for a significant length of time. A transitory state is one in which the object upon entering, immediately exits to some other state. You should model transitory states if there is any significant behavior of the object during entry and exit.

### Step 3 – Specify the Partial Ordering of States

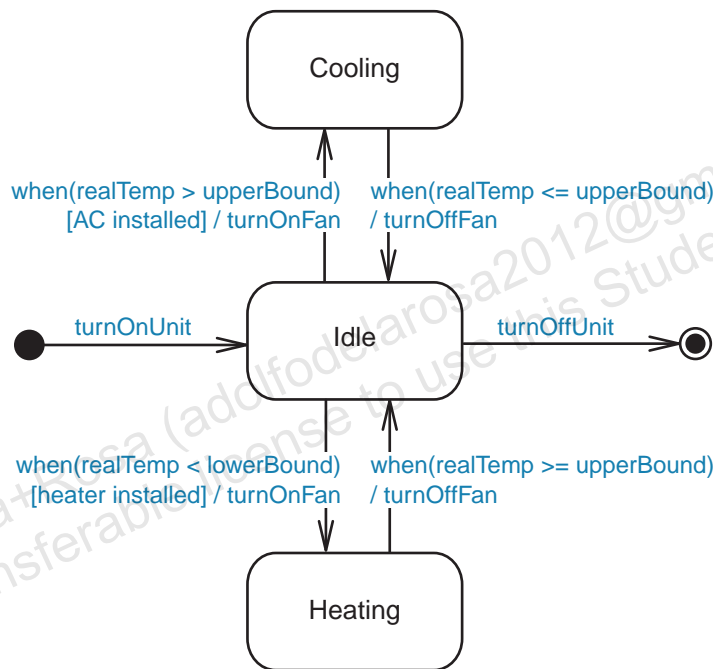
Then specify the transitions between the object states; this is known as a partial ordering of the states. For example, an HVAC object can transition from the Idle state to the Cooling or Heating states, but the Cooling state cannot transition to the Heating state. Figure 9-8 illustrates this.



**Figure 9-8** Step 3 – Specify the Partial Ordering of States

## Step 4 – Specify the Transition Events and Actions

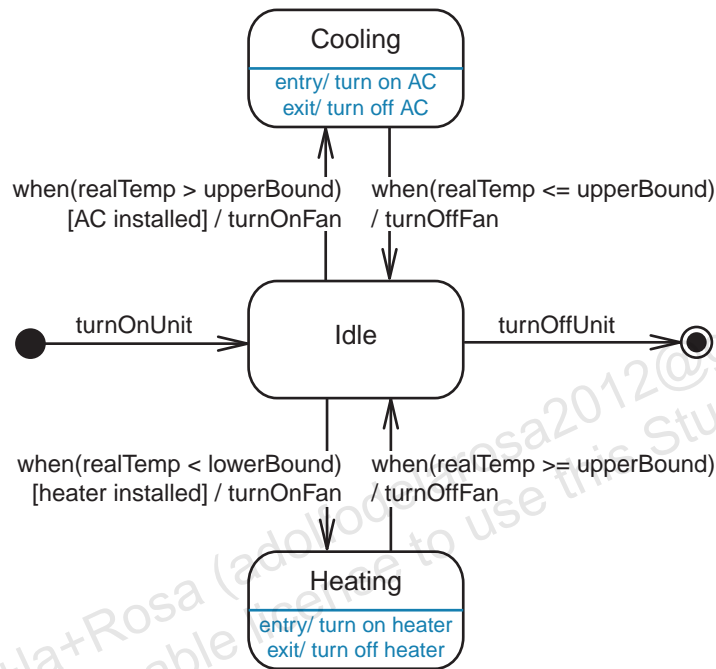
Then specify the triggers, guard conditions, and actions (if any) for each transition. If no trigger is specified, then the transition occurs immediately (unless guarded). If no guard condition is specified, then the transition occurs when the trigger occurs. The guard condition can halt a state transition even if the trigger occurs. If no actions are specified, then the transition occurs without affecting the object (other than changing state). If actions are specified, then these actions act upon the object before the new state is entered. Figure 9-9 illustrates this step of the HVAC state model.



**Figure 9-9** Step 4 – Specify the Transition Events and Actions

## Step 5 – Specify the Actions Within a State

Finally, specify any events and corresponding actions that occur within each state (if any). For example, the Cooling state indicates that the AC subsystem is turned on after this state is entered, and the AC is turned off after the state is exited. Figure 9-10 illustrates this step of the HVAC state model.



**Figure 9-10** Step 5 – Specify the Actions Within a State

## After Trigger Event

Events that occur after a period of time are shown by using the *after* trigger event.

To fire a transition after 10 minutes, specify the event on the transition as `after(10 mins)`.

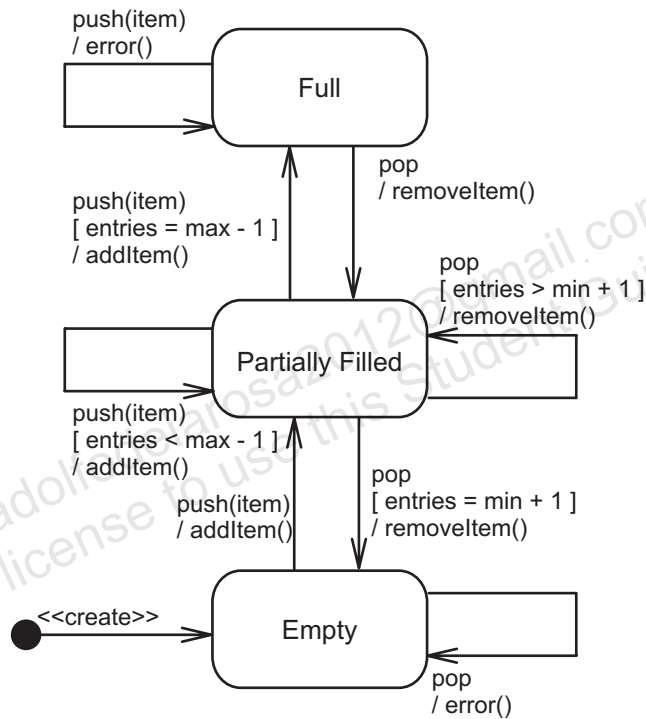
This event is often used for timeouts.



## Self Transition

A self transition is a state transition with the same state for the source and the destination of the transition. As a result of self transition, the exit and entry internal actions also fire.

Figure 9-11 shows an example Stack in which the push and pop trigger events transition to another state or transition back to the same state.

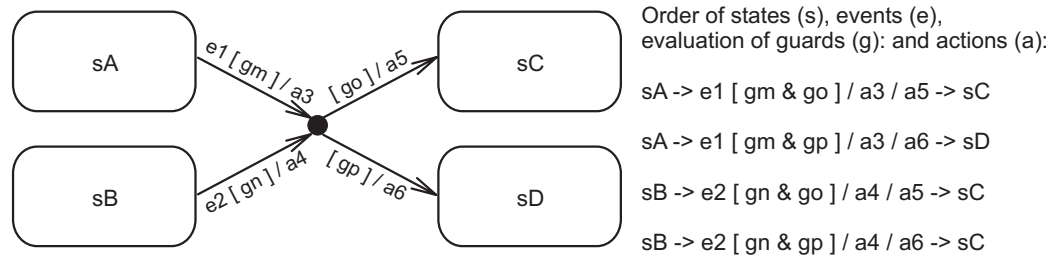


**Figure 9-11** Stack Example Showing Self Transition

## Junction

In UML, a Junction is used to simplify diagrams by breaking the transition into several fragments, thereby reducing the duplication of trigger events, guards and actions.

Figure 9-12 shows the order of state transitions, order of events, evaluation of guards, and actions.



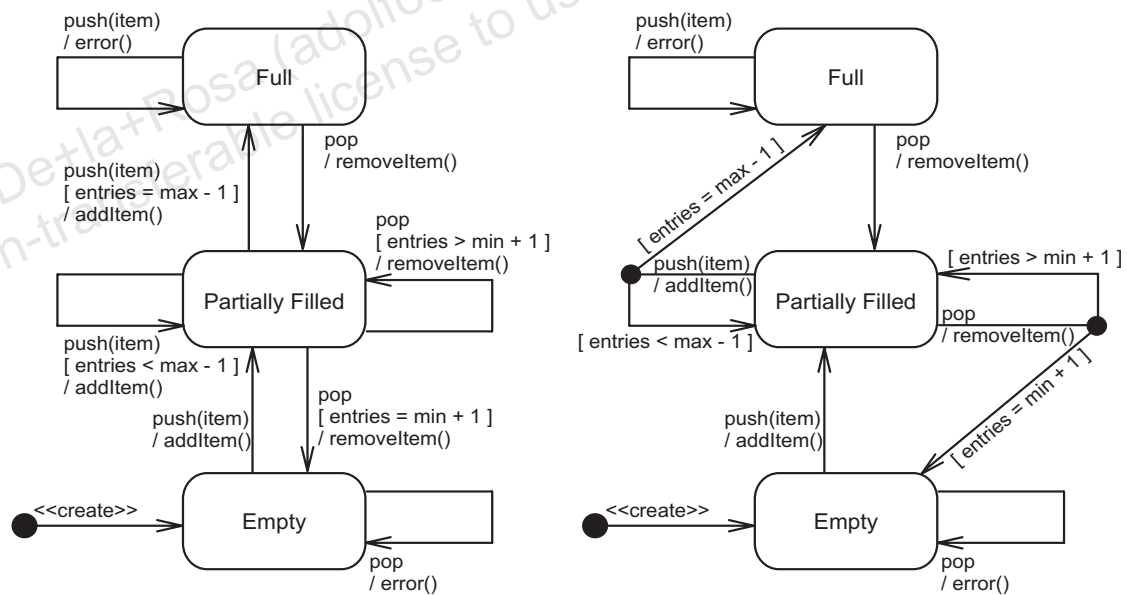
**Figure 9-12** Order of Processing by Using Junction

## Junctions Example

Figure 9-13 shows two examples of a stack. One example uses the original state transitions notation. The other example shows the advantages of

Stack example without using a junction

Equivalent stack example using a junction



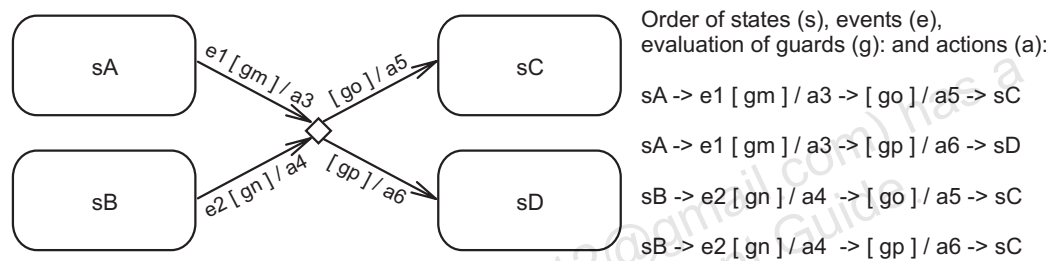
using a junction in which the number of duplicate trigger events and actions are reduced.

**Figure 9-13** Comparing Stacks With Junction and Without Junction

## Choice

In UML, a Choice is used to simplify diagrams by breaking the transition into several fragments, thereby reducing the duplication of trigger events, guards, and actions. Choice also enables dynamic evaluation of the guards after the previous transition fragments actions are executed.

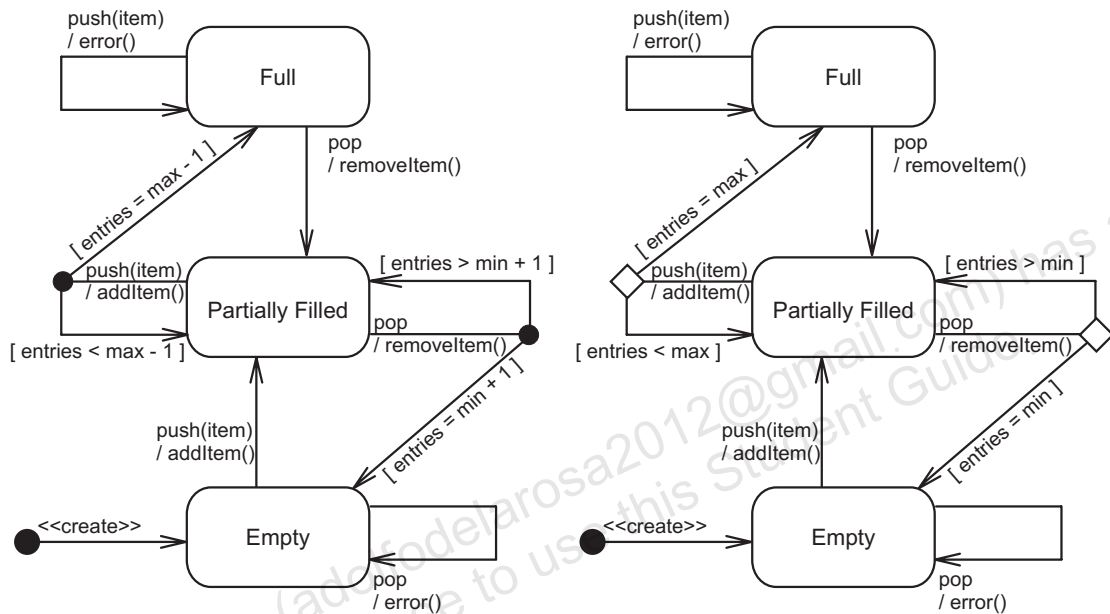
Figure 9-14 shows the order of state transitions, order of events, evaluation of guards, and actions.



**Figure 9-14** Order of Processing by Using Choice

## Choice Example

Figure 9-15 compares two examples of a stack. One example uses the Junction state transitions notation and the other example shows the differences by using a choice state transition notation.



**Figure 9-15** Comparing Stacks With Choice and Without Choice

## Summary

In this module, you were introduced to the UML State Machine diagram and the concepts behind objects with a complex state. Here are the important concepts:

- A object might have states that define unique behaviors for the object.
- The State Machine diagram provides a mechanism for modeling the states and transitions of an object.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a  
non-transferable license to use this Student Guide.

# Applying Design Patterns to the Design Model

---

## Objectives

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

- Define the essential elements of a software pattern
- Describe the Composite pattern
- Describe the Strategy pattern
- Describe the Observer pattern
- Describe the Abstract Factory pattern
- Describe the State pattern

## Additional Resources



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

- Alexander, Christopher. *A Pattern Language: Towns Buildings Construction*. Oxford University Press, Inc., 1977.
- 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.
- Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.
- Metske, Steven John. *Design Patterns Java Workbook*. Addison Wesley Professional, 2002
- Meyer, Bertrand. *Object-Oriented Software Construction (2nd ed)*. Upper Saddle River: Prentice Hall, 1997.
- Shalloway, Alan, and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, 2001.
- Stelting, Stephen, and Olav Maassen. *Applied Java Patterns*. Palo Alto: Sun Microsystems Press, 2002.
- Vlissides, John, James Coplien, and Norman Kerth. *Pattern Language of Program Design (vol. 2)*. Reading: Addison Wesley Longman, Inc., 1996.



---

**Note** – There are also several good Web sites on Software Patterns:

<http://hillside.net/patterns> and

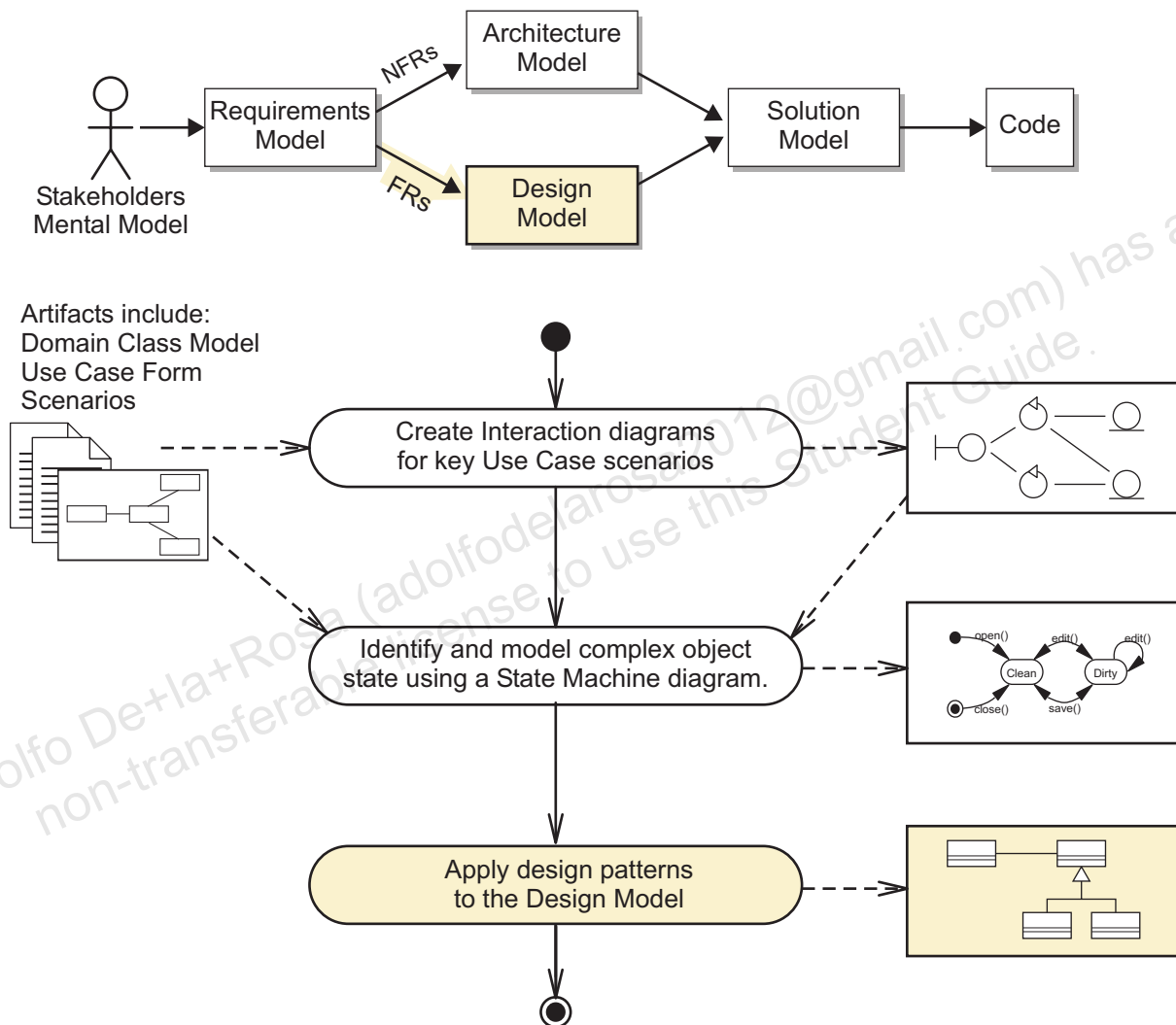
<http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html>

---



# Process Map

This module describes the next step in the Design workflow: applying design patterns to the Design model. Figure 10-1 shows the activity and artifacts discussed in this module.



**Figure 10-1** Design Workflow Process Map

# Explaining Software Patterns

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” (Alexander page x)

This statement was written about building architecture, but it is also true in object-oriented design. In software design, patterns appear over and over again.

A *software pattern* is a “description of communicating objects and classes that are customized to solve a general design problem in a particular context.” (Gamma, Helm, Johnson, Vlissides page 3)



---

**Note** – The authors of the *Design Patterns* book Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides are commonly referred to collectively as the Gang of Four (GoF).

---

The discipline of software patterns formalizes these patterns by identifying four essential elements of the pattern:

- **Pattern name**  
This element provides a convenient and memorable term for the pattern. This element is useful when the development team communicates about the software design because it raises the level of abstraction in the conversation.
- **Problem**  
This element states the conditions under which the pattern applies.
- **Solution**  
This element states the structural and dynamic behavior of the components that make up the pattern.
- **Consequences**  
This element states the results of the pattern as well as any trade-offs incurred by applying the pattern.



**Note** – There are potentially dozens of elements that could be described about pattern; for example, the GoF use twelve elements in their pattern descriptions. However, the previous four elements are considered essential.

## Levels of Software Patterns

Large software systems can be visualized at many levels of depth. Similarly, software patterns can be applied at many levels of depth. Patterns can be roughly grouped into the following three levels:

- Architectural patterns

Patterns at this level manifest at the highest software and hardware structures within the system. For example, a Remote Proxy (GoF) is a pattern that enables a client component to access a remote service as if the client were accessing a local service. An RMI stub is an example of a Remote Proxy.

Architectural patterns usually support the non-functional requirements of the system. For example, the Remote Proxy can increase the throughput of the Client (or Presentation) tier by moving an expensive service to another hardware node.

- Design patterns

Patterns at this level manifest at the mid-level software structures within the system. These patterns apply to a small set of classes or components.

Design patterns usually support the functional requirements of the system. For example, the Composite pattern directly supports a FR which states “the system contains a set of objects that all support the operations of X, Y, and Z; furthermore, the group of these object must also support those operations.”

- Idioms

Patterns at this level manifest at the lowest software structures (classes and methods) within the system. These patterns apply either within a class definition or even with the code of a method. For example, Localized Ownership (Vlissides, Coplien, and Kerth page 5) is a pattern for managing object creation and destruction in C++.

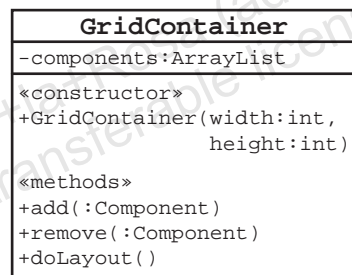
Idioms usually support language-specific features. For example, the Localized Ownership pattern supports the memory management model of C++ programs; it is not applicable to Java technology because the JVM software supplies automated memory management using garbage collection.

## Design Principles

There are several design principles that support the solutions of software patterns:

- Open Closed Principle (OCP)
- Composite Reuse Principle (CRP)
- Dependency Inversion Principle (DIP)

The following sections discuss each of these principles using a common example. This example describes the design of a GUI `GridContainer` class; a GUI container that places each GUI component element in a grid starting from the top left and adding components to the right and to the bottom. Figure 10-2 shows this component and an calculator example.



```
GridContainer calcGUI = new GridContainer(4,4);
calcGUI.add(new Button("1"));
calcGUI.add(new Button("2"));
calcGUI.add(new Button("3"));
calcGUI.add(new Button("+"));
calcGUI.add(new Button("4"));
calcGUI.add(new Button("5"));
calcGUI.add(new Button("6"));
calcGUI.add(new Button("-"));
```

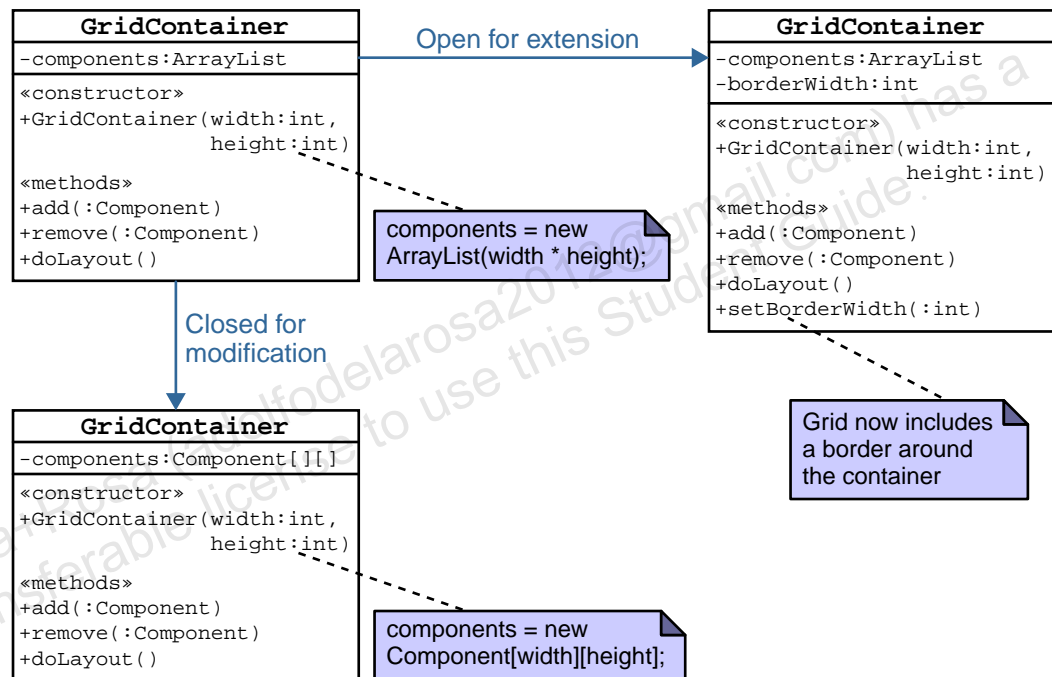
1	2	3	+
4	5	6	-
7	8	9	*
0	.	=	/

**Figure 10-2** GUI Example that Demonstrates Design Principles

## Open Closed Principle

“Classes should be open for extension but closed for modification.”  
(Knoernschild page 8)<sup>1</sup>

The point of this principle is that you should be able to change a class without affecting the clients of that class. There are two dimensions to this kind of change, change by extending the capabilities of the class and change by modifying the implementation of the class. Figure 10-3 illustrates this principle.



**Figure 10-3** Example of the Open Closed Principle

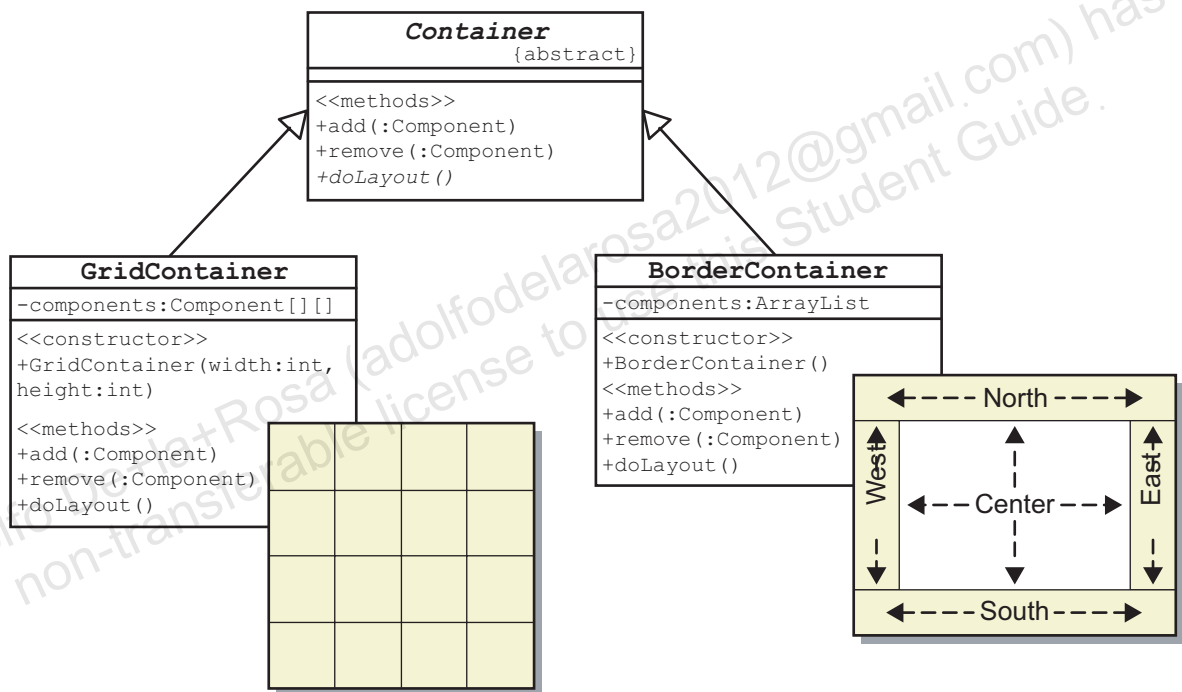
The GridContainer class can be extended by adding a line border around the grid. This extension is made by adding the setBorderWidth method. An existing client of this class does not need to be changed because this behavior does not change the original behavior of the class. Also, the GridContainer class can be modified because the private implementation of the layout mechanism is hidden from the client. In particular, the data structure of the components attribute can be changed from a linear list to a matrix. An existing client of this class need not be changed due to this implementation change because the interface has not changed.

1. This principle was first introduced by Bertrand Meyer in OOSC page 57. However, Knoernschild's treatment is easier to understand.

## Composite Reuse Principle

“Favor polymorphic composition of objects over inheritance.”  
(Knoernschild page 17)<sup>2</sup>

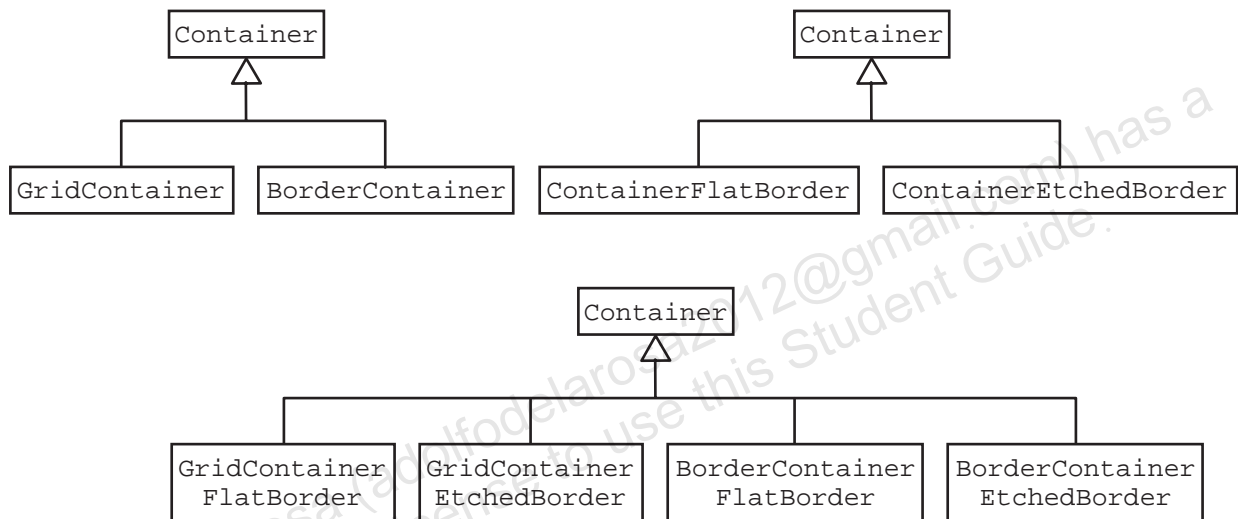
The point of this principle is that creating new behaviors by using inheritance is often not very flexible. In the GUI container example, you might need a new type of container layout: instead of a grid, you might want to layout a group of components into regions around the container screen. You could create an abstract Container superclass with two subclasses, GridContainer and BorderContainer. Figure 10-4 illustrates this.



**Figure 10-4** Using Inheritance to Provide Different GUI Layout Mechanisms

2. CRP was first introduced in the GoF *Design Patterns* book.

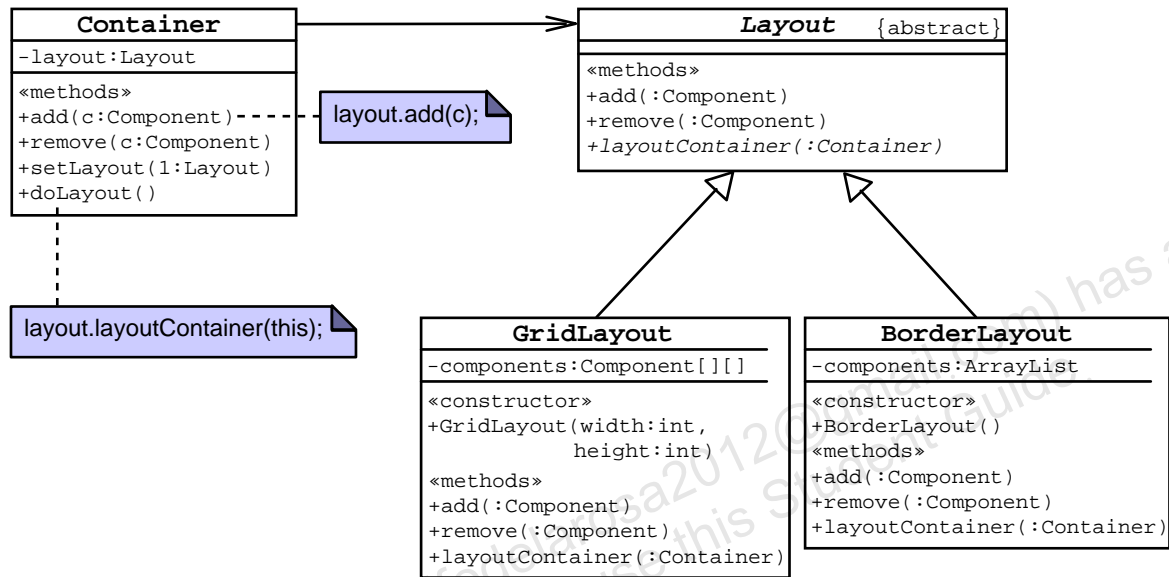
There is nothing inherently wrong with this strategy. However, GUI containers can have many different features. Suppose that you want to be able to create a border around the group of GUI components. You could create a flat border which draws a line around the boundary of the GUI components. Or you could draw an etched border around the components to give a 3D effect to the container. Conceptually, you have two independent hierarchies of functionality, one for container layout and one for drawing a border around the container on the screen. However, you cannot have two hierarchies, but rather a single hierarchy that is a cross product of the two features. Figure 10-5 illustrates this.



**Figure 10-5** Variation in Multiple Features Leads to Combinatorial Explosion of Classes

This is a brittle class structure. Suppose that you now needed a new border visualization (for example, a rounded corner border). You would have to create two additional classes: `GridContainerRoundedBorder` and `BorderContainerRoundedBorder`. If you have  $N$  types of layout and  $M$  types of borders, then you potentially must create  $N * M$  container classes.

CRP provides an alternative to this problem. For each feature of a GUI container, you could create a separate class hierarchy that supports only that feature. The Container class would then use an implementation of a class that supports that feature. For example, you could have a single Container class that delegates the layout functionality to an independent object. Figure 10-6 illustrates this.



**Figure 10-6** Example of the Composite Reuse Principle

The GUI border functionality can also be delegated by the Container class. This is not shown in the previous diagram, but it does make sense to use CRP to delegate this functionality to a separate object, with a class hierarchy independent of the Container class.

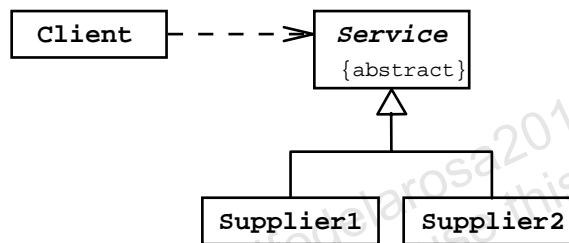


## Dependency Inversion Principle

“Depend on abstractions. Do not depend on concretions.”  
(Knoernschild page 12)<sup>3</sup>

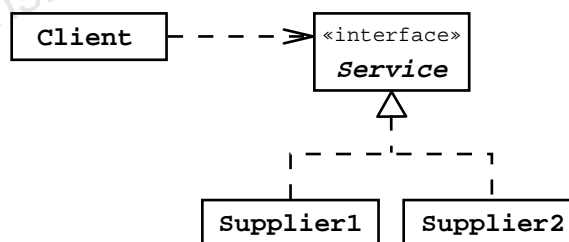
CRP provides a glimpse of the Dependency Inversion Principle. The point of DIP is that if you need to delegate to another component, then it is best to write the client code to an abstract class. Figure 10-6 on page 10-10 shows an example of DIP. The Container class uses an object of the Layout class hierarchy. The Container class does not need to know which layout manager is used, because the client of the container uses the `setLayout` method to specify the layout manager at runtime.

Figure 10-7 illustrates a generic version of this principle.



**Figure 10-7** Dependency Inversion Principle Using an Abstract Class

Figure 10-8 shows DIP can be used with interfaces.



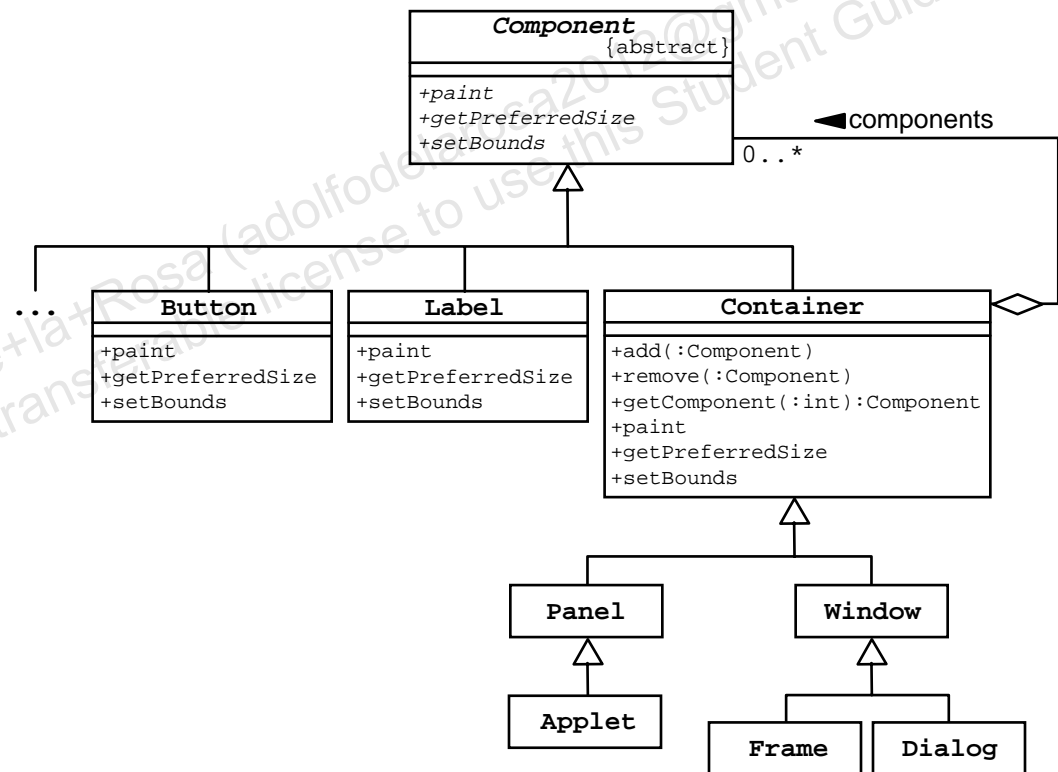
**Figure 10-8** Dependency Inversion Principle Using an Interface

3. DIP was originally introduced by Robert Martin in a C++ *Report* article. This article is available online at <http://www.objectmentor.com/resources/articles/dip.pdf>

## Describing the Composite Pattern

“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” (GoF page 163)

The Composite pattern deals with whole-part hierarchies. A GUI container is an example of a whole-part hierarchy: the container holds one or more GUI components. For example, a screen might have a group of GUI components to collect customer information, name, address, and so on. The screen (implemented by a Frame class) is the container and labels and text fields are the parts that make up the whole. Abstract Window Toolkit (AWT) includes a class called Container that enables the programmer to add and remove components to the container. The Frame class is a subclass of Container. Figure 10-9 illustrates this class hierarchy.



**Figure 10-9** An Example of the Composite Pattern in AWT

## Composite Pattern: Problem

The Composite pattern solves a problem having the following characteristics:

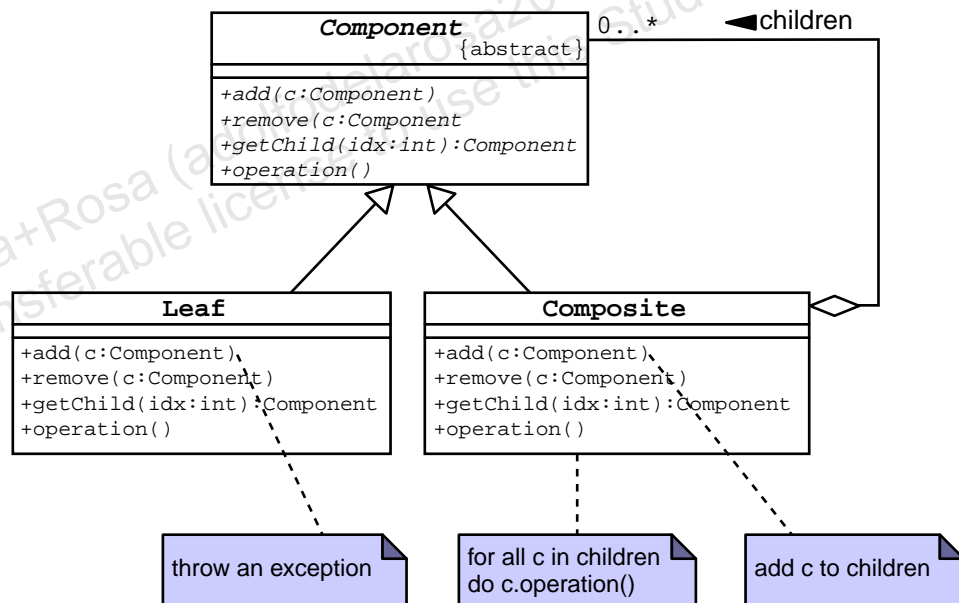
- You want to represent whole-part hierarchies of objects
- You want to use the same interface on the assemblies and the components in an assembly

## Composite Pattern: Solution

The solution for the Composite pattern has the following characteristics:

- Create an abstract class, `Component`, that acts as the superclass for concrete “leaf” and `Composite` classes.
- The `Composite` class can be treated as a component because it supports the `Component` class interface.

Figure 10-10 illustrates the Composite pattern solution described by the GoF. In this solution, the `Component` class includes all of the composite methods: `add`, `remove`, and `getChild`.



**Figure 10-10** GoF Solution for the Composite Pattern

This might seem confusing at first because leaf classes would not have meaningful implementations of these methods. However, this solution provides the most flexibility for the client code because all subclasses of Component use the same interface. An alternate solution puts the composite methods only in the Composite class. Figure 10-11 illustrates this.

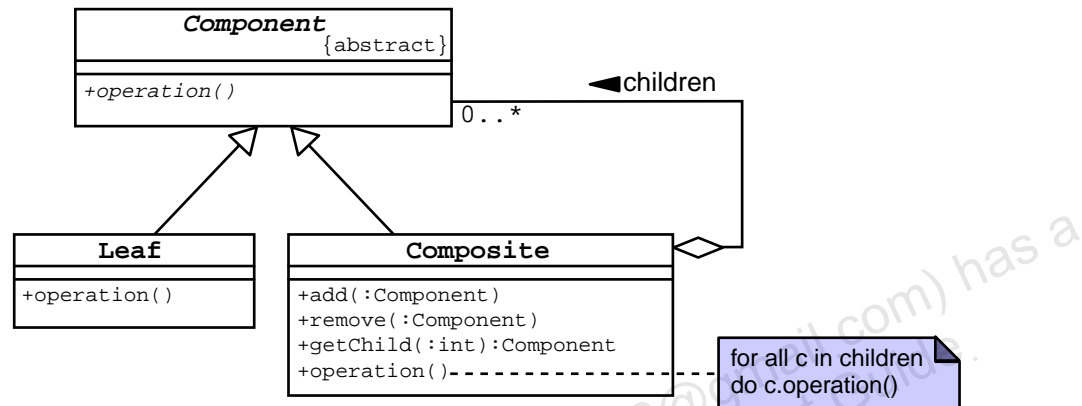


Figure 10-11 An Alternate Solution for the Composite Pattern

## Composite Pattern: Consequences

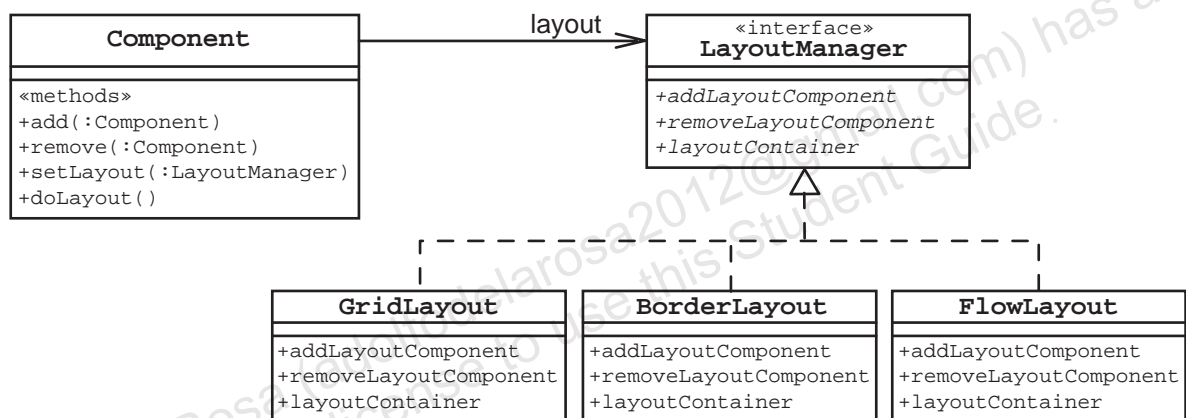
The consequences of the Composite pattern include:

- Makes the client simple  
The client of a composite parts hierarchy does not need to care what type of component (including a composite component) it is dealing with; it can call any operation on the component object.
- Makes it easier to add new kinds of components  
You can add new types of components (and even new types of composites) to the component hierarchy. As long as the new class follows the same interface, then the client classes do not have to be modified.
- Can make the design model too general  
This pattern makes the component hierarchy very generic. A composite can contain *any type* of component. There might be cases in which you want to restrict the types of components a given composite can hold. This is not possible with this pattern.

## Describing the Strategy Pattern

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” (GoF page 315)

GUI containers in AWT can be organized into several predefined layouts: grid, border, flow, and so on. AWT provides this mechanism by using a layout manager that is assigned to every GUI container. The container delegates the algorithm of laying out its components to this layout manager. This is a direct implementation of the Strategy pattern. Figure 10-12 illustrates this AWT example.



**Figure 10-12** An Example of the Strategy Pattern in AWT

## Strategy Pattern: Problem

The Strategy pattern solves a problem having the following characteristics:

- You have a set of classes that are only different in the algorithms that they use
- You want to change algorithms at runtime

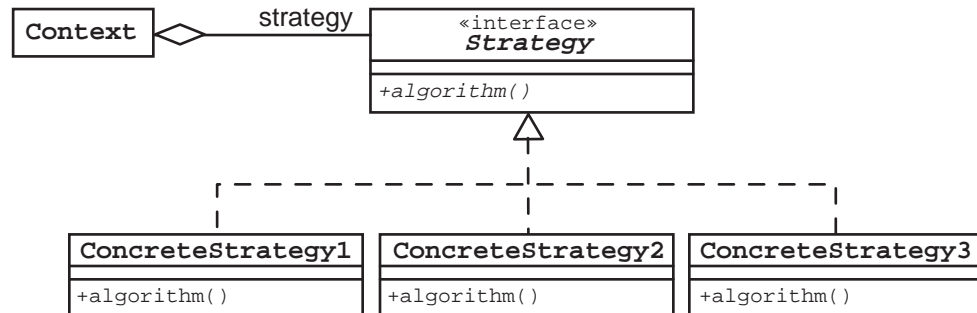
## Strategy Pattern: Solution

The solution for the Strategy pattern has the following characteristics:

- Create an interface, Strategy, that is implemented by a set of concrete “algorithm” classes.

- At runtime, select an instance of these concrete classes within the Context class.

Figure 10-13 illustrates the Strategy pattern solution described by the GoF.



**Figure 10-13** GoF Solution for the Strategy Pattern

## Strategy Pattern: Consequences

The consequences of the Strategy pattern include:

- An alternate to subclassing  
As you saw with GUI containers in “Composite Reuse Principle” on page 10-8, it might seem reasonable to subclass a Context (such as a GUI container) into subclasses that supports specific algorithms, such as GUI layouts. This technique leads to an artificial hierarchy of classes (see Figure 10-4 on page 10-8) or a combinatorial explosion of classes if there are multiple algorithms that need to vary (see Figure 10-5 on page 10-9). The Strategy pattern eliminates the need to subclass to support multiple algorithms.
- Strategies eliminate conditional statements  
Some programmers might consolidate multiple algorithms into a single algorithm that uses conditional statements to branch between the differences in each algorithm. This solution leads to complex code that is hard to maintain. The Strategy pattern eliminates conditional statements by putting each algorithm in its own class.
- A choice of implementations  
Using the Strategy pattern, the context object can change the algorithm at runtime. This solution increases the flexibility of the software.
- Communication overhead between Strategy and Context patterns

On the downside, the Strategy pattern introduces additional communication overhead between the strategy objects and the context object. This could lead to complicated code if a given strategy requires private details of the context.

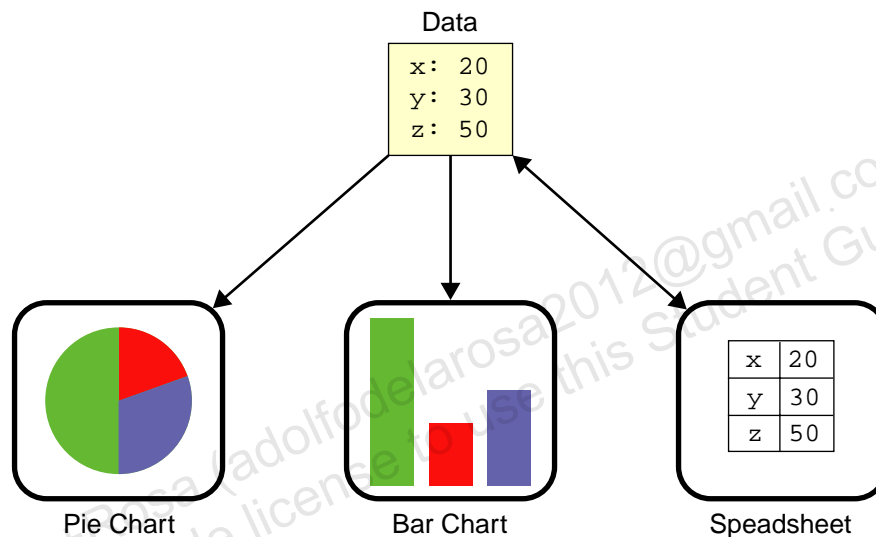
- Increased number of objects

Using the Strategy pattern introduces the need for additional objects (strategy objects) at runtime. However, these objects might not require any attributes of their own; therefore, you could use a single instance of each concrete strategy class that all contexts share.

## Describing the Observer Pattern

“Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically.” (GoF page 293)

Suppose that you have a set of data (a collection of numbers) that you need to visualize using several different techniques using a pie chart, bar chart, or as a row in a spreadsheet. Figure 10-14 illustrates this situation.

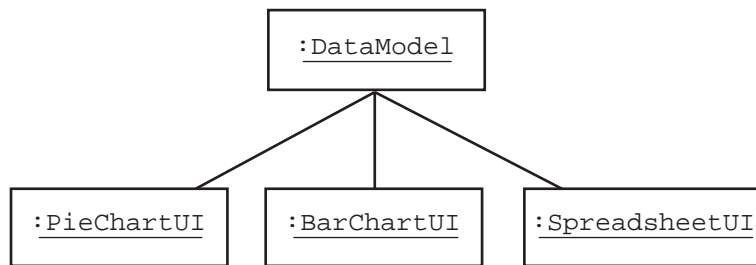


**Figure 10-14** An Example of the Observer Pattern in a GUI Application

If you kept a copy of the data in each GUI component, then your software would have to modify the data in each of these components when the data changed in the system. This tight coupling between the UI components and the data model is brittle. A change in the data model would require changes to each UI component that must visualize that data.

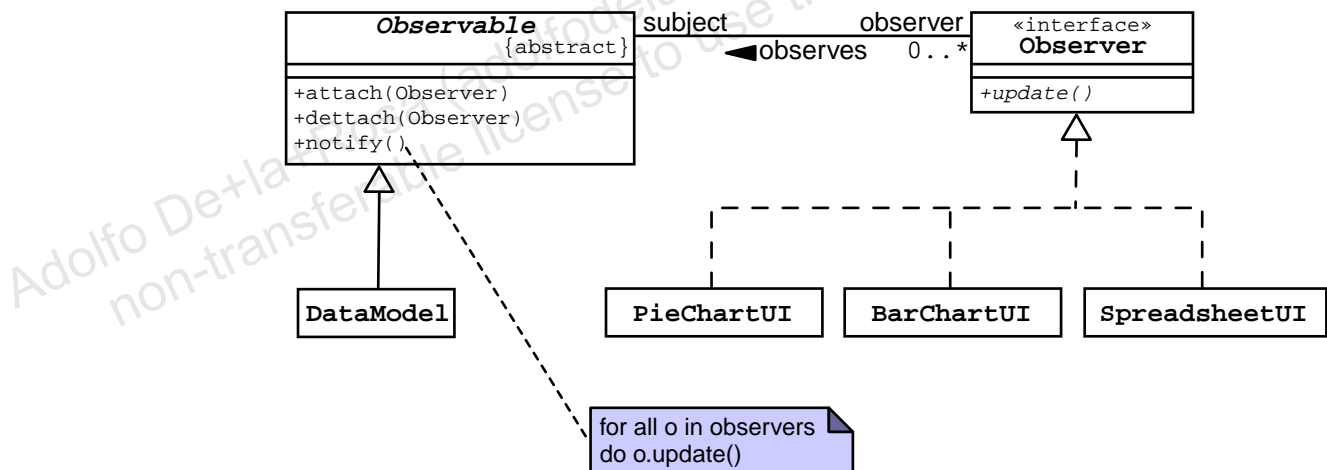


Alternatively, you could keep the data in a separate object and then notify the UI components when that data has changed. Figure 10-15 illustrates this alternate approach in an Object diagram.



**Figure 10-15** Object Diagram of the Observer Example

The UI classes must implement the Observer interface to perform the necessary changes to the UI representation when the update method is invoked. The DataModel class inherits from the Observable abstract class which provides the methods to attach and detach observers. The DataModel class is responsible for invoking the notify method whenever its data change. Figure 10-16 illustrates this design.



**Figure 10-16** Class Diagram of the Observer Example

## Observer Pattern: Problem

The Observer pattern solves a problem having the following characteristics:

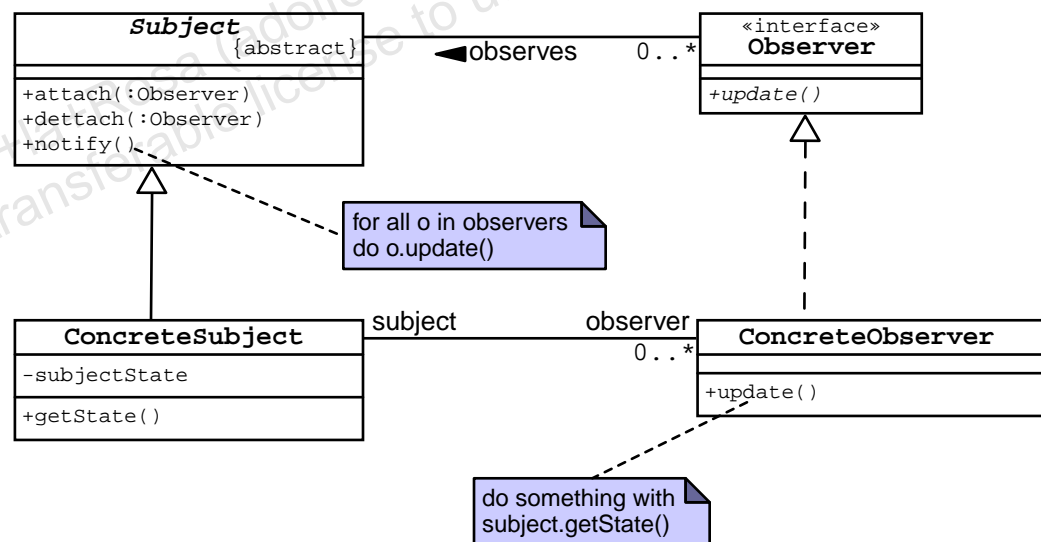
- You need to notify a set of objects that an event has occurred.
- The set of observing objects can change at runtime.

## Observer Pattern: Solution

The solution for the Observer pattern has the following characteristics:

- Create an abstract class Subject that maintains a collection of Observer objects.
- When a change occurs in a subject, it notifies all of the observers in its set.

Figure 10-17 illustrates the Observer pattern solution described by the GoF.



**Figure 10-17** GoF Solution for the Observer Pattern

## Observer Pattern: Consequences

The consequences of the Observer pattern include:

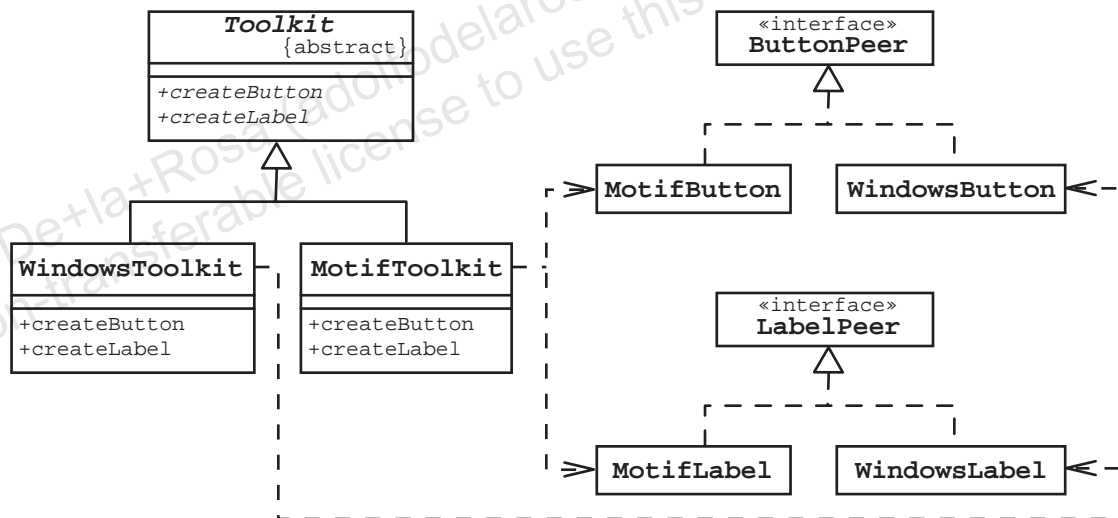
- Abstract coupling between Subject and Observer  
The coupling between the subject object and the observer object is abstract. This coupling is abstract because the subject does not know the concrete class of each observer; it only knows that the observer object supports the Observer interface (the update method).
- Support for multicast communication  
Observer objects can dynamically attach to a subject. Whenever the subject changes, all attached observers are notified in a broadcast of update messages to every Observer.
- Unexpected updates  
A downside to this pattern is that the communication mechanism is coarse-grained. Observers might not care if one part of the data model is changed, but there is no mechanism for the subject to know which Observer is interested in which data changes. It is up to the Observer to query the data model and decide if the changes need to be processed.

Java technology uses the Observer pattern in several different ways. In AWT, each GUI component can listen for user input events, such as typing a character in a text component, selecting a menu item, or clicking on a button. You can create and attach listener objects to GUI components that observe various user events for that component. These listeners are observers in the Observer pattern and the GUI component is the subject.

## Describing the Abstract Factory Pattern

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.” (GoF page 87)

Java technology is based on using the Java virtual machine to provide platform-independence. Creating platform-independent GUIs was particularly challenging. The design of the AWT required the separation of abstract UI components, such as `java.awt.Button`, from the actual operating system components that physically presented the UI component on the screen. The real components are called peers of the abstract UI components. Therefore, there is a set of peers for UI components on the Motif platform as well as a set of peers on the Microsoft Windows platform. The abstract AWT component uses the AWT Toolkit class to create the appropriate peer at runtime. When the J2SE platform Software Development Kit (SDK) is created for a specific platform, then the SDK includes a concrete subclass of Toolkit which provides the methods to generate the relevant peers for that platform. Figure 10-18 illustrates this design.



**Figure 10-18** An Example of the Abstract Factory Pattern in AWT

**Note** – Swing is the second generation GUI framework for Java technology that provides true platform-independence. Swing GUI components are not based on peers.



## Abstract Factory Pattern: Problem

The Abstract Factory pattern solves a problem having the following characteristics:

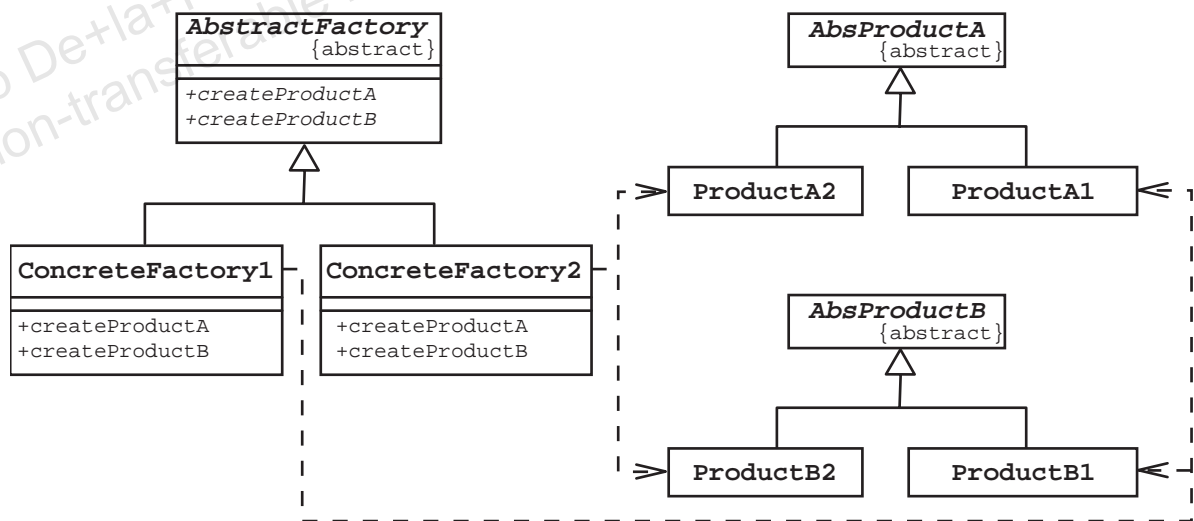
- A system has multiple families of products.
- Each product family is designed to be used together.
- You do not want to reveal the implementation classes of the product families.

## Abstract Factory Pattern: Solution

The solution for the Abstract Factory pattern has the following characteristics:

- Create an abstract creator class that has a factory method for each type of product.
- Create a concrete creator class that implements each factory method which returns a concrete product.

Figure 10-19 illustrates the Abstract Factory pattern solution described by the GoF.



**Figure 10-19** GoF Solution for the Abstract Factory Pattern

## Abstract Factory Pattern: Consequences

The consequences of the Abstract Factory pattern include:

- Isolates concrete classes  
This pattern hides the concrete product classes, so the clients of these products must use the abstract class (or interface) thus enforcing the client into using the DIP.
- Makes exchanging product families easy  
You can add and remove whole product families without changing the client code, because it depends on the abstract product classes rather than on the concrete classes.
- Promotes consistency among products  
“When product objects in a family are designed to work together, it is important that an application use objects from only one family at a time. Abstract Factory makes this easy to enforce.” (GoF page 90)  
The best example of this in Java technology is a variation on Abstract Factory in the JDBC framework. The factory methods start at the DriverManager class level. The `getConnection(URL)` method returns a `Connection` object for the specific JDBC driver that matches the URL. The `Connection` interface supplies the `createStatement` factory method. This method returns a `Statement` object containing a `executeQuery` method that creates a `ResultSet` object. In the JDBC framework, the factory methods are scattered among multiple interfaces: `Connection`, `Statement`, and `ResultSet`. These are the products for which every JDBC vendor must provide an implementation (a concrete product). These products are built by the vendor to work together; the Abstract Factory variation in the JDBC framework is built to enforce that consistency.
- Supporting new kinds of products is difficult  
On the downside, changing the set of products is rather difficult because you have to update all of the product families that are currently used.

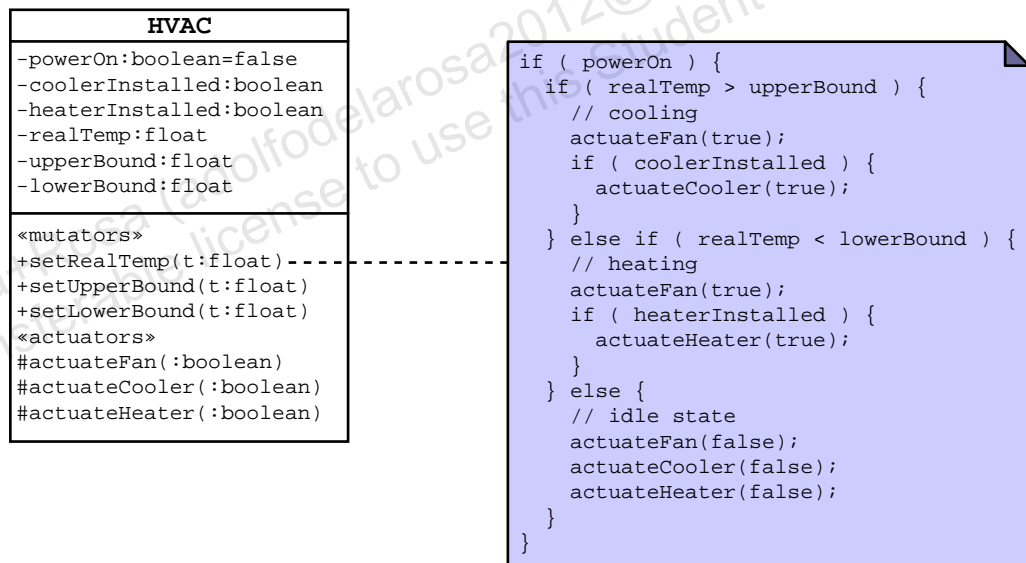
# Programming a Complex Object

This section discusses the problems with programming a complex object and suggests a design pattern to simplify this development.

## Problems With Coding a Complex Object

Before introducing the State pattern, you should understand why the pattern exists at all. Objects with complex state often are very difficult to code. A naive approach is to use the attributes of the object to determine the state of the object which performs the actions of the state. This can lead to very complex, and hard to maintain code because such code usually includes significantly complex conditional statements.

Figure 10-20 shows an example implementation of the `setRealTemp` method in the HVAC class.



**Figure 10-20** HVAC Class With Complex State Code

Not only is the code complex, but the code might not truly capture the state behavior of the object. The `setRealTemp` method has several if and if-else constructs to determine if the HVAC object should be cooling, heating, or idle. In the else-clause for the idle condition, the method calls three actuator methods. If these methods are expensive operations, then this code is wasting time being idle. To code this method correctly would require even more conditional logic.

Another issue is: How do you change this code if a new state is introduced to the HVAC system? Creating new states or changing the behavior of existing states can be a maintenance nightmare because you will have to review *all* of the code in the HVAC class.

The State pattern solves all of these problems.

## Describing the State Pattern

“Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.” (GoF page 305)

The State pattern introduces a set of state objects within which the behavior of the primary object is encapsulated in the state class methods. For the HVAC class, you would create four state classes: *InitialState*, *IdleState*, *CoolingState*, and *HeatingState*. Each of these classes implements a *State* interface that is specifically designed for the primary object (each primary class has its own interface for its state objects).

The primary class keeps a state attribute that holds the particular state object for the state of the primary object. The primary class includes a protected *setState* method which performs the state transitions for the primary object. This method calls the exit action before changing state, changes the state attribute, and then calls the entry actions for the new state. The state-dependent methods of the primary class delegate the behavior to the current state object. In the HVAC class, the *setRealTemp* method stores the new temperature value and then calls the corresponding method on the state object. Figure 10-21 illustrates the HVAC class using the State pattern.

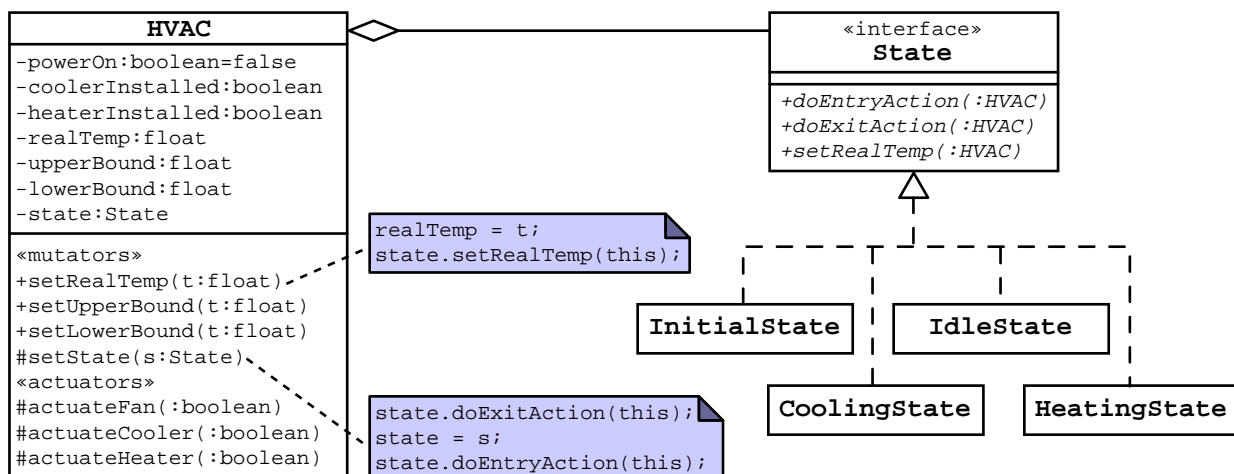
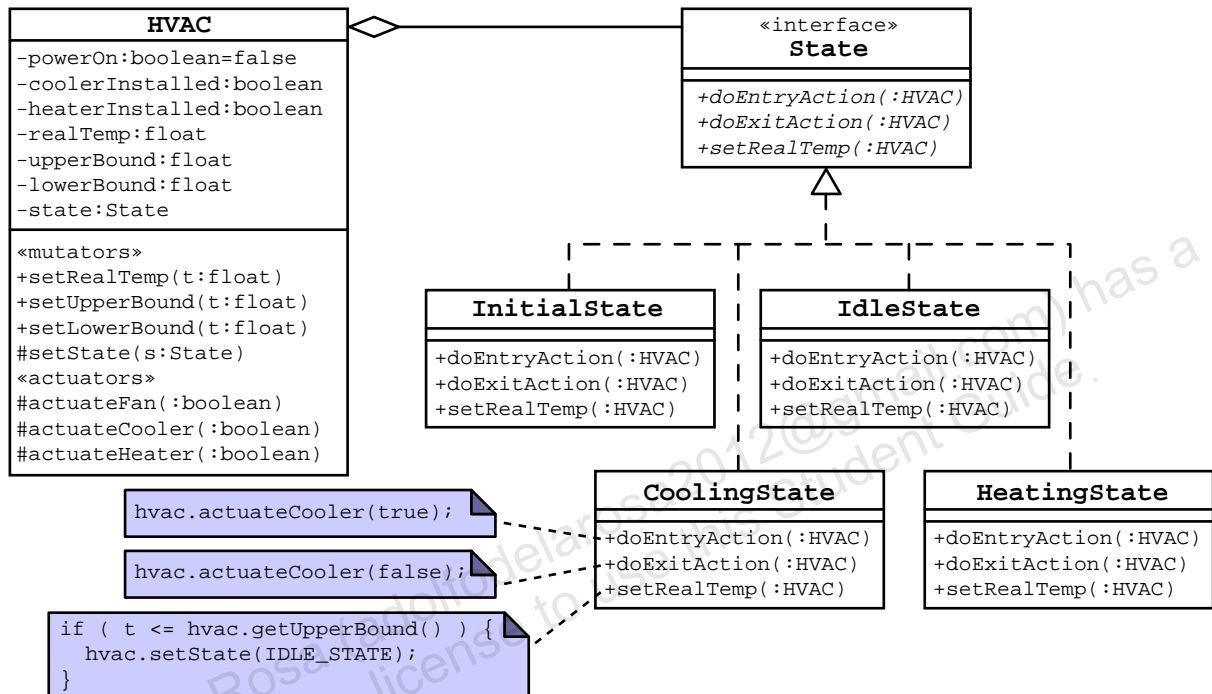


Figure 10-21 HVAC Class Using the State Pattern



Each state class focuses only on the behavior of that state. For example, the `CoolingState` class includes an entry action of turning on the AC subsystem, an exit action of turning the AC off, and a `setRealTemp` method that changes the state of the HVAC object to `Idle` if the temperature has dropped below the upper bound value. Figure 10-22 illustrates this.



**Figure 10-22** HVAC Class Using the State Pattern

## State Pattern: Problem

The State pattern solves the problem having the following characteristics:

- An object's runtime behavior depends on its state.
- Operations have large, multipart conditional statements that depend on the object's state.

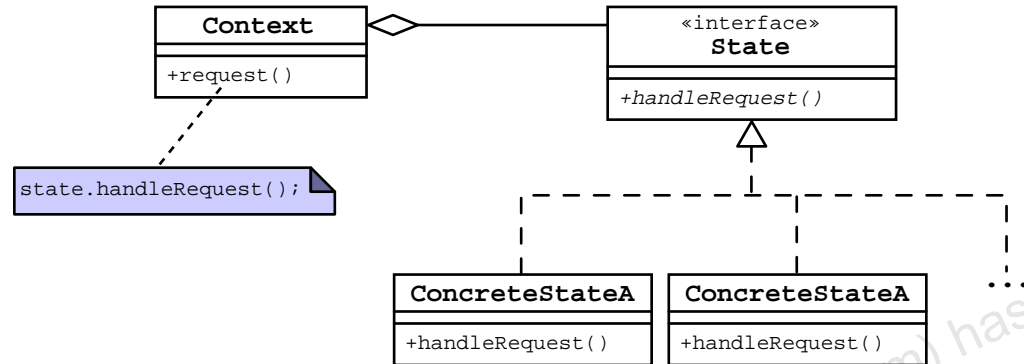
## State Pattern: Solution

The solution for the State pattern has the following characteristics:

- Create an interface that specifies the state-based behaviors of the object.
- Create a concrete implementation of this interface for each state of the object.

- Dispatch the state-based behaviors of the object to the object's current state object.

Figure 10-23 illustrates the State pattern solution described by the GoF.



**Figure 10-23** GoF Solution for the State Pattern

## State Pattern: Consequences

The consequences of the State pattern include:

- Localizes state-specific behavior  
State-specific behavior is localized to the class that models that particular state. This solution greatly simplifies the development of state objects. The State pattern easily permits the addition of new state classes or the modification of the behavior of existing states without affecting the code in the other state classes.
- Reduces conditional statements  
Without using the State pattern, the methods of the primary class tend to use complex conditional code to keep track of the internal state of the object. The State pattern greatly reduces this type of conditional code.
- Makes state transitions explicit  
Without using the State pattern, the methods of the primary class often hide the details of when the object transitions from one state to another. Using the State pattern makes the state of the object and its state transitions explicit.
- Increased the number of objects  
Using the State pattern introduces the need for additional objects (state objects) at runtime. However, these objects might not require any attributes of their own; in that case, you could use a single instance of each concrete state class that all objects share.
- Communication overhead between the State and Context objects

The State pattern introduces additional communication overhead between the state objects and the context object. This could lead to complicated code if a given state requires private details of the context. One solution to this problem is to make the state classes as inner classes of the context class.

## Summary

In this module, you were introduced to a few classic Design Patterns. Here are a few important concepts:

- Software patterns provide proven solutions to common problems.
- Design principles provide tools to build and recognize software patterns.
- Patterns are often used together to build more flexible and robust systems and frameworks (such as AWT and JDBC).

# 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

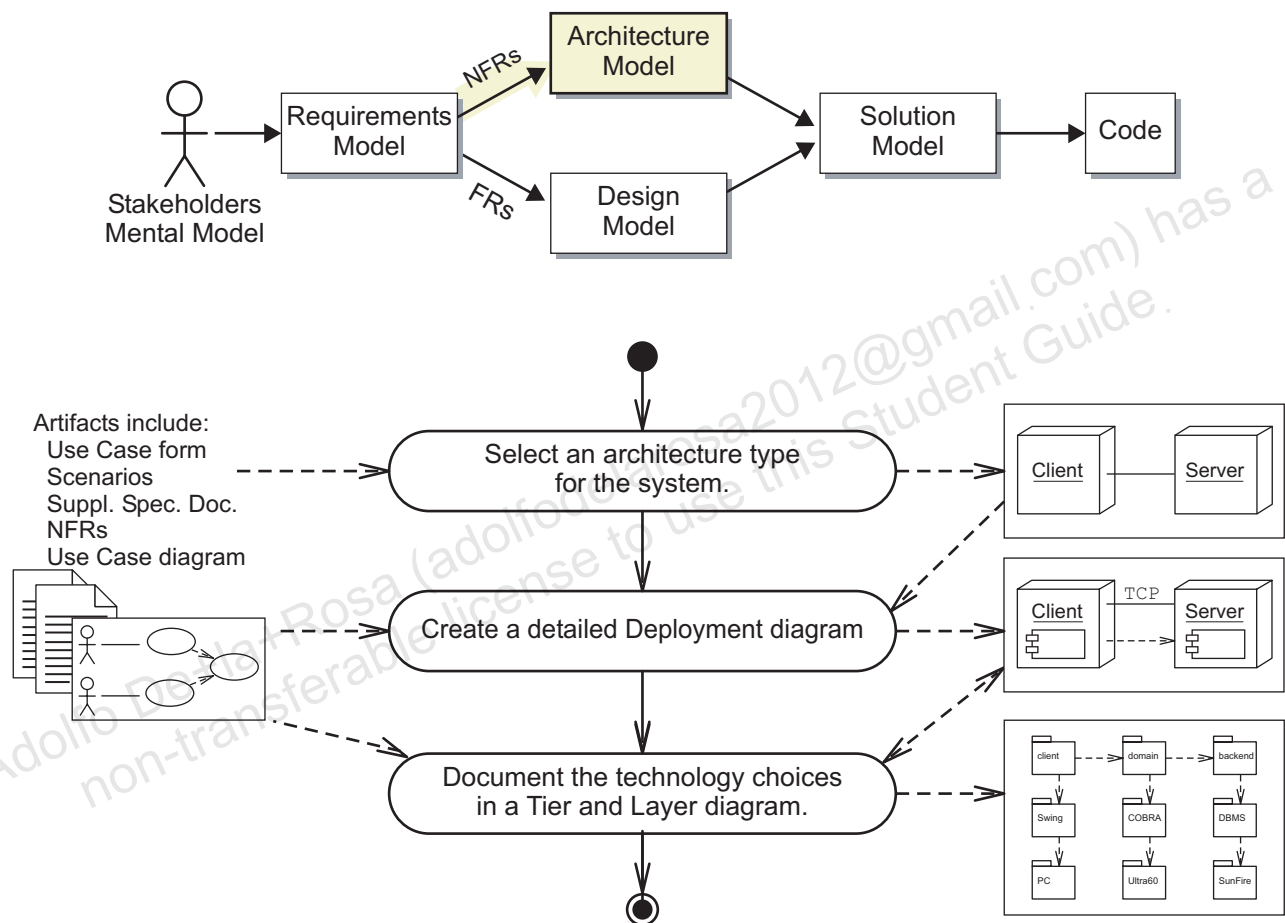


**Additional resources** – The following references provide additional information on the topics described in this module:

- 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.html>].

# 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.

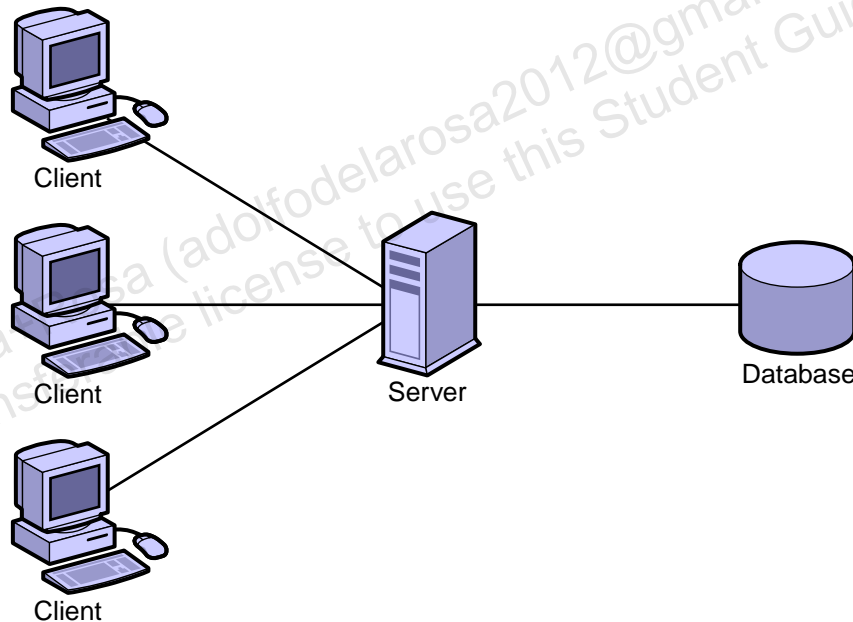


## Risks Associated With Large-Scale, Distributed Enterprise Systems

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 diagram illustrates a multi-tier architecture with three tiers of servers, firewalls, clients, and databases. The components are arranged as follows:

- Top Tier:** A **Database** (cylinder icon) is connected to a **Services** server (tower icon) via a bidirectional arrow.
- Second Tier:** The **Services** server is connected to a **Firewall** (brick wall icon) via a bidirectional arrow. The **Firewall** is connected to two **Server** units (tower icons) via bidirectional arrows.
- Third Tier:** The **Firewall** is connected to a **Firewall** (brick wall icon) via a bidirectional arrow. This **Firewall** is connected to three **Tiers** (tower icons) via bidirectional arrows. The **Tiers** are connected to a **Database** (cylinder icon) via bidirectional arrows.
- Client Tier:** Three **Client** units (desktop computer icons) are connected to the **Firewall** in the second tier via bidirectional arrows.

The diagram shows a complex network of connections between these components, representing a multi-tier architecture.

**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



## 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<sup>SM</sup> Solutions services group. Table 11-1 shows how architects differ from designers.

**Table 11-1** Difference Between Architects and Designers

	<b>Architect</b>	<b>Designer</b>
<b>Abstraction level</b>	High/broad Focus on few details	Low/specific Focus on many details
<b>Deliverables</b>	System and subsystem plans, architectural prototype	Component designs, code specifications
<b>Area of focus</b>	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

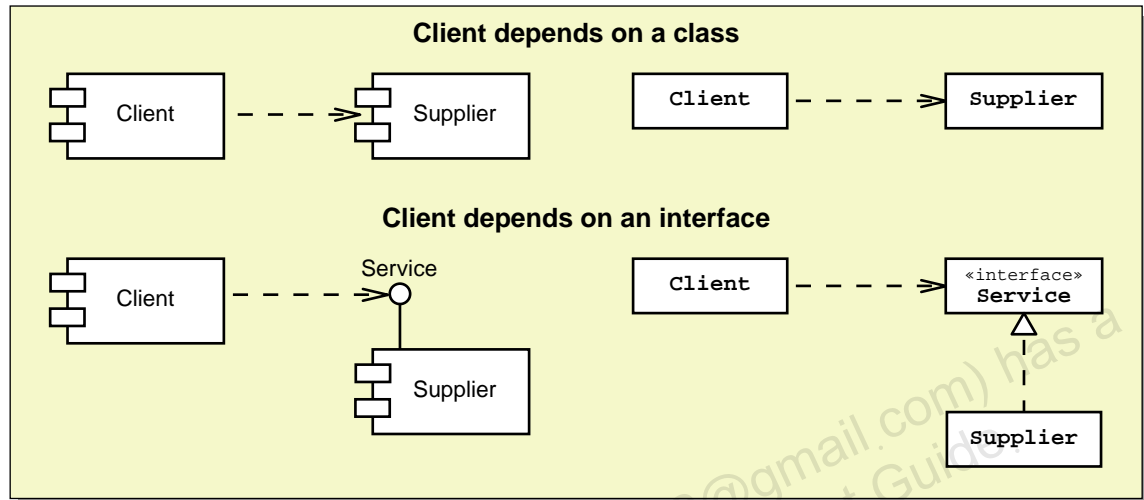
Architecture principles are axioms or assumptions that suggest good practices when constructing an system architecture. These principles form the basis of many architectural patterns (see “Architectural Patterns and Design Patterns” on page 11-14) and form the decisions an architect makes to satisfy 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<sup>®</sup> 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.

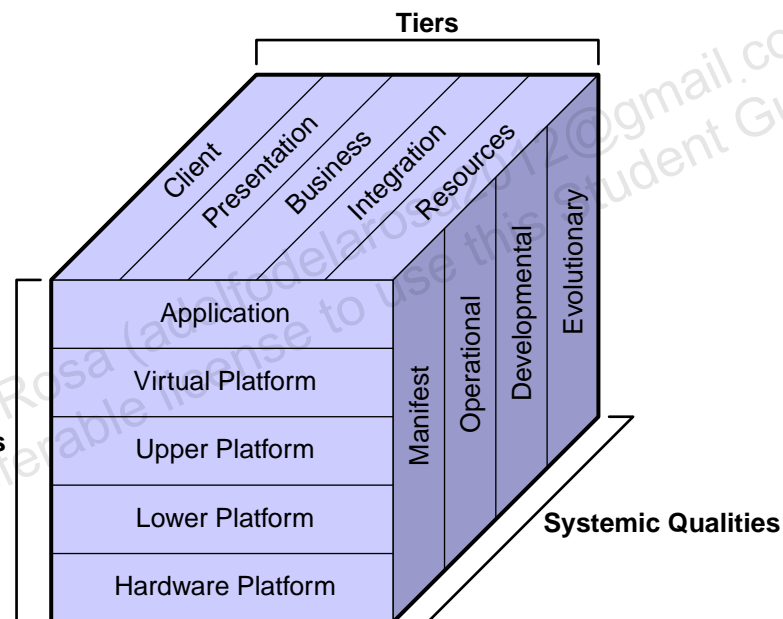
---

# 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.

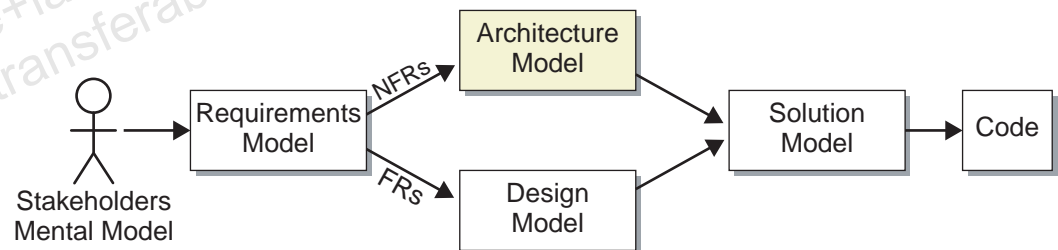
Architecture is a process of selecting trade-offs. Some systemic qualities will be more important than others. The architect must make the appropriate decisions that determine which systemic qualities to focus on.

## 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<sup>1</sup> 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.

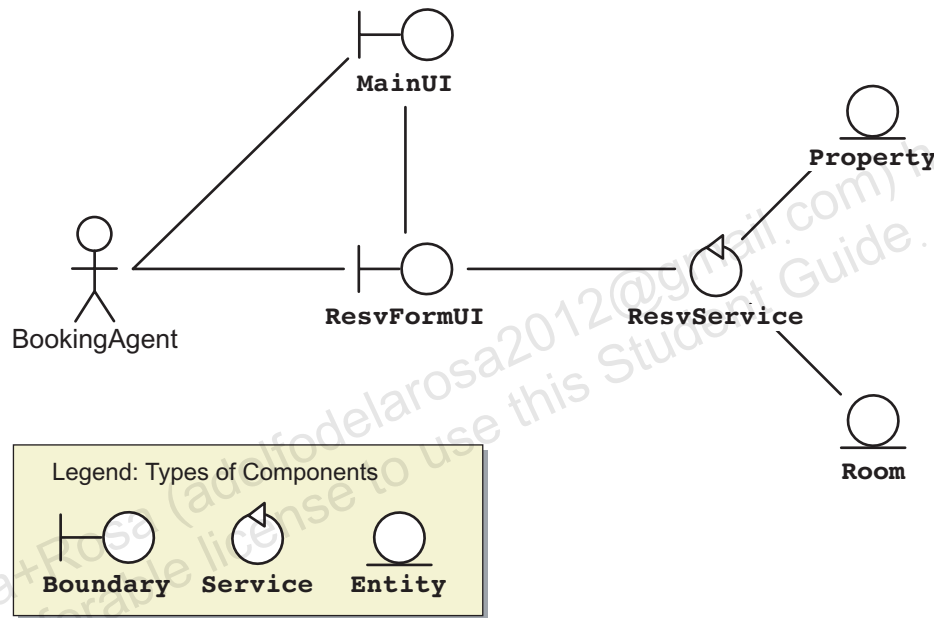
This is not a formal artifact of the Architecture workflow. However, you will use this template later in the course. This step is discussed in “Creating the Architecture Template” on page 11-47.

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.



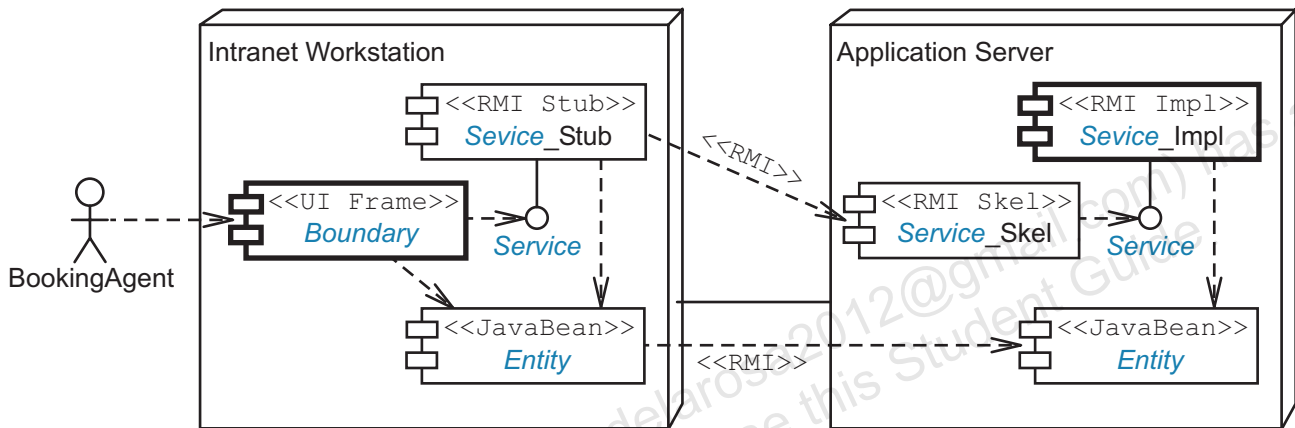
**Figure 11-7** Example Design Components

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.

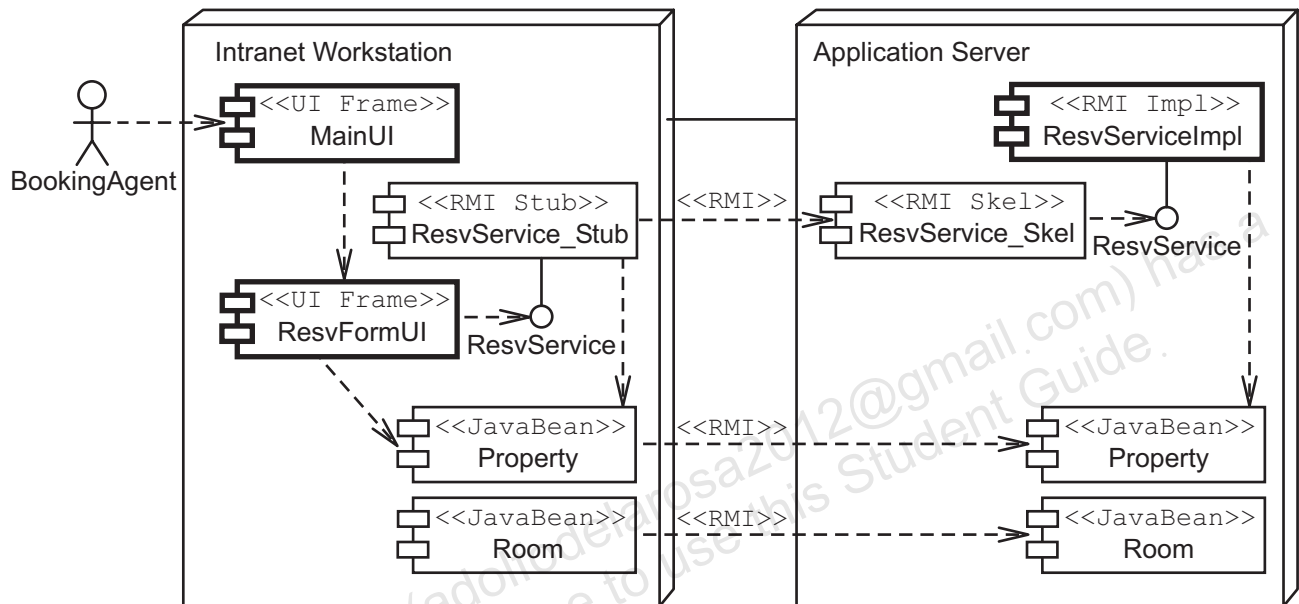


**Figure 11-8** Example Architecture Template

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.

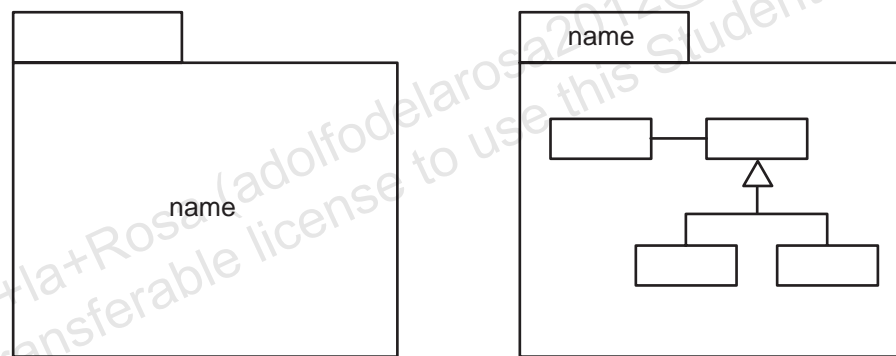
## Describing Key Architecture Diagrams

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.



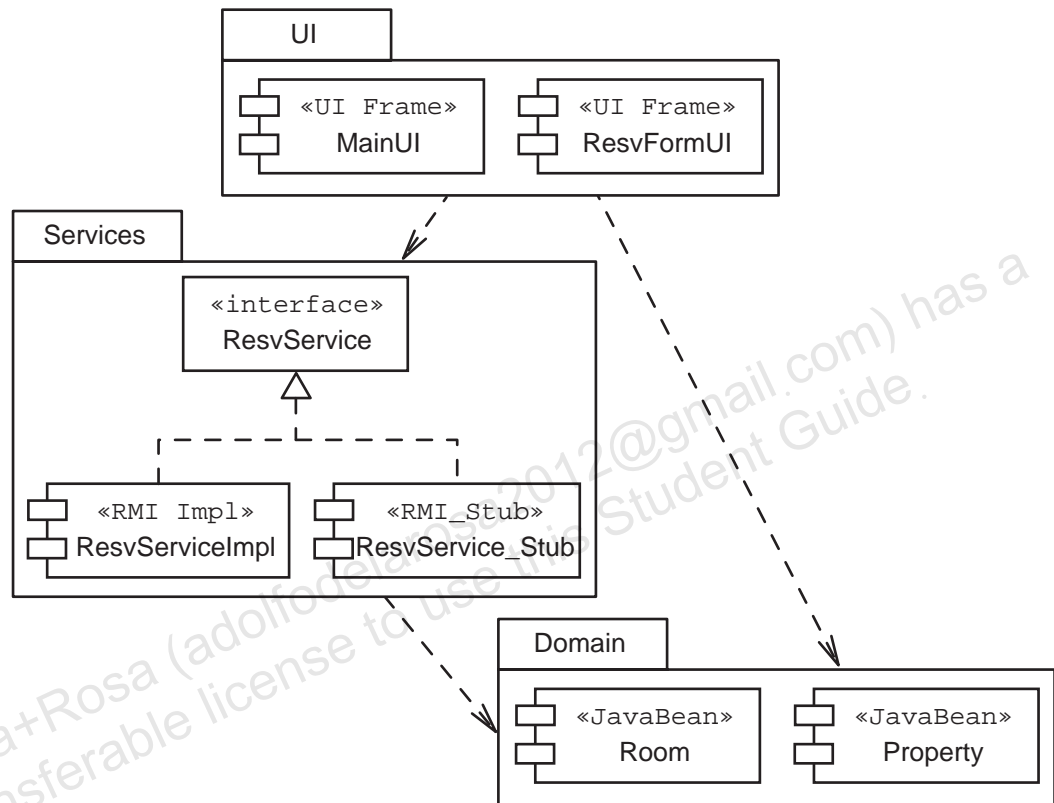
**Figure 11-10** Elements of a UML Package Diagram

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.

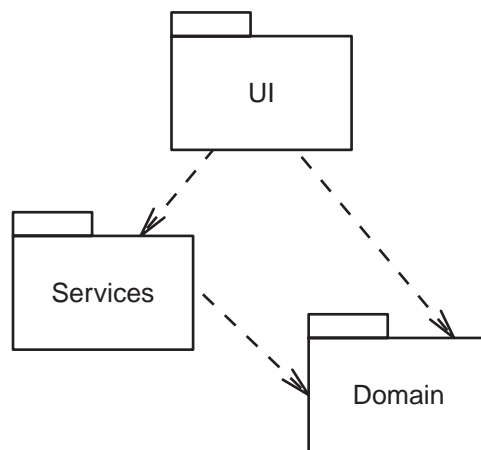


For example, you can group your UI components into a UI package, your services into the Services package, and your entities into a Domain package. The UI components depend on the business services and domain entities, and the service components depend on the domain entities. Figure 11-11 shows these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

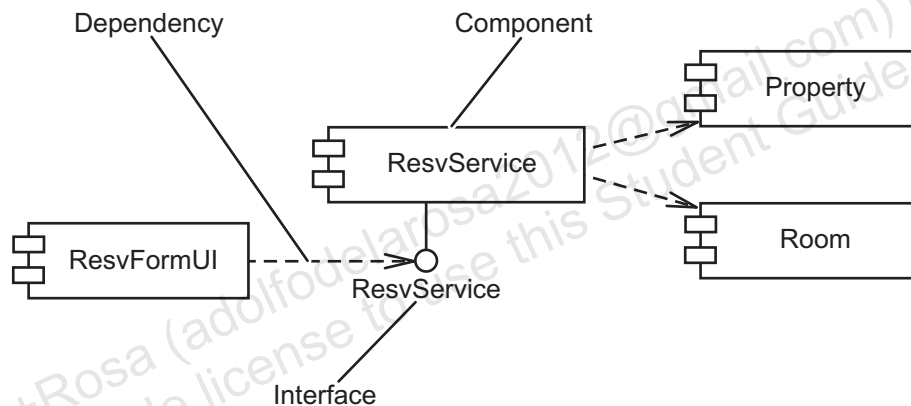


## Identifying the Elements of a Component Diagram

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

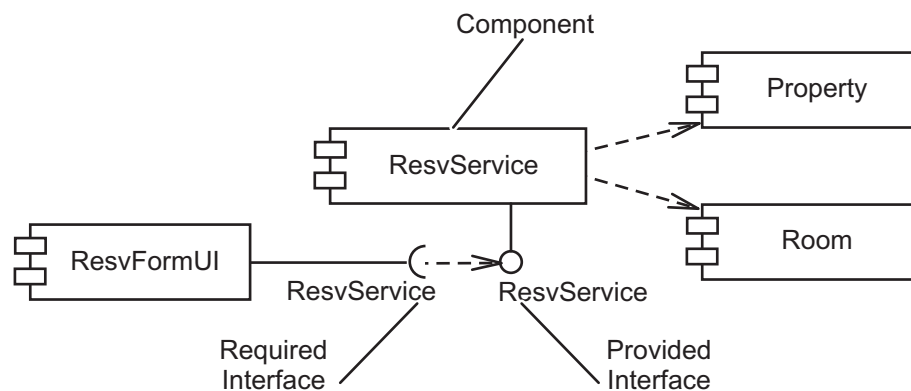
### Purpose of a Component Diagram

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.



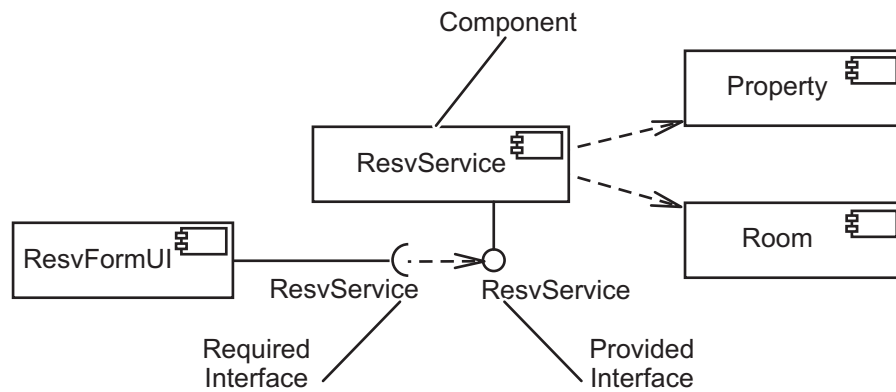
**Figure 11-13** An Example Component Diagram

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

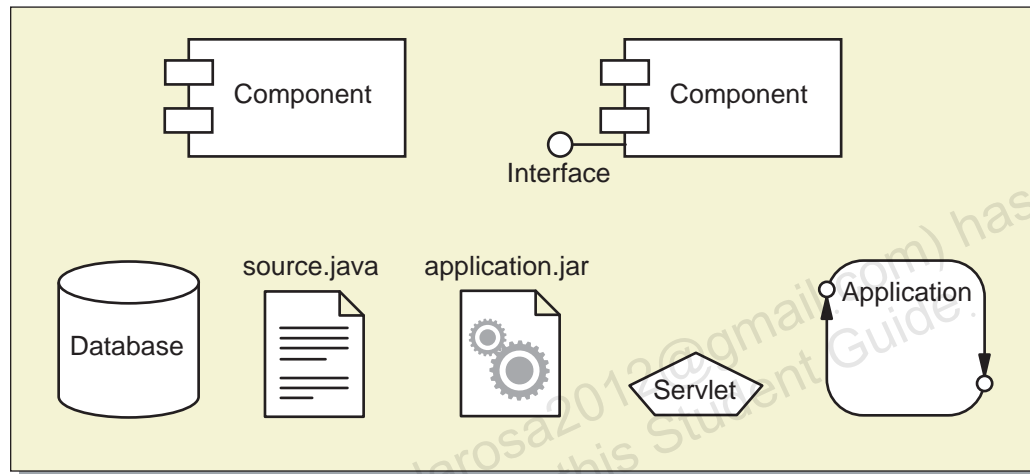
## Characteristics of a Component

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.

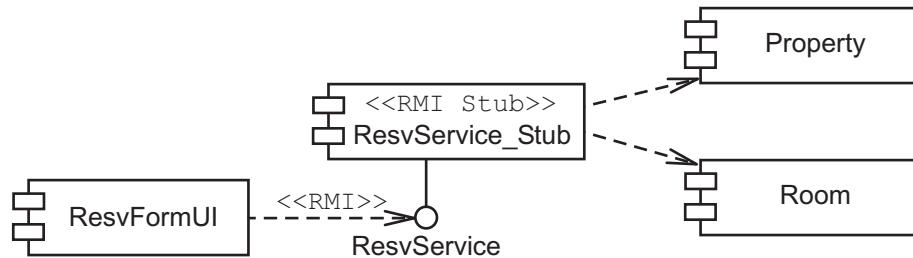


**Figure 11-16** Types of Software Components

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.

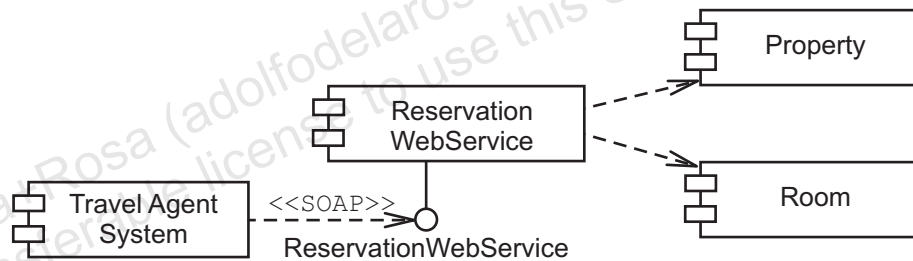
## Example Component Diagrams

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.



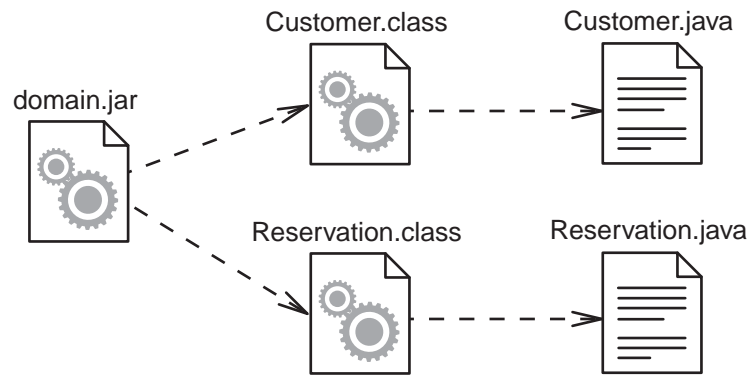
**Figure 11-17** Component Diagrams Can Show Software Dependencies

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.

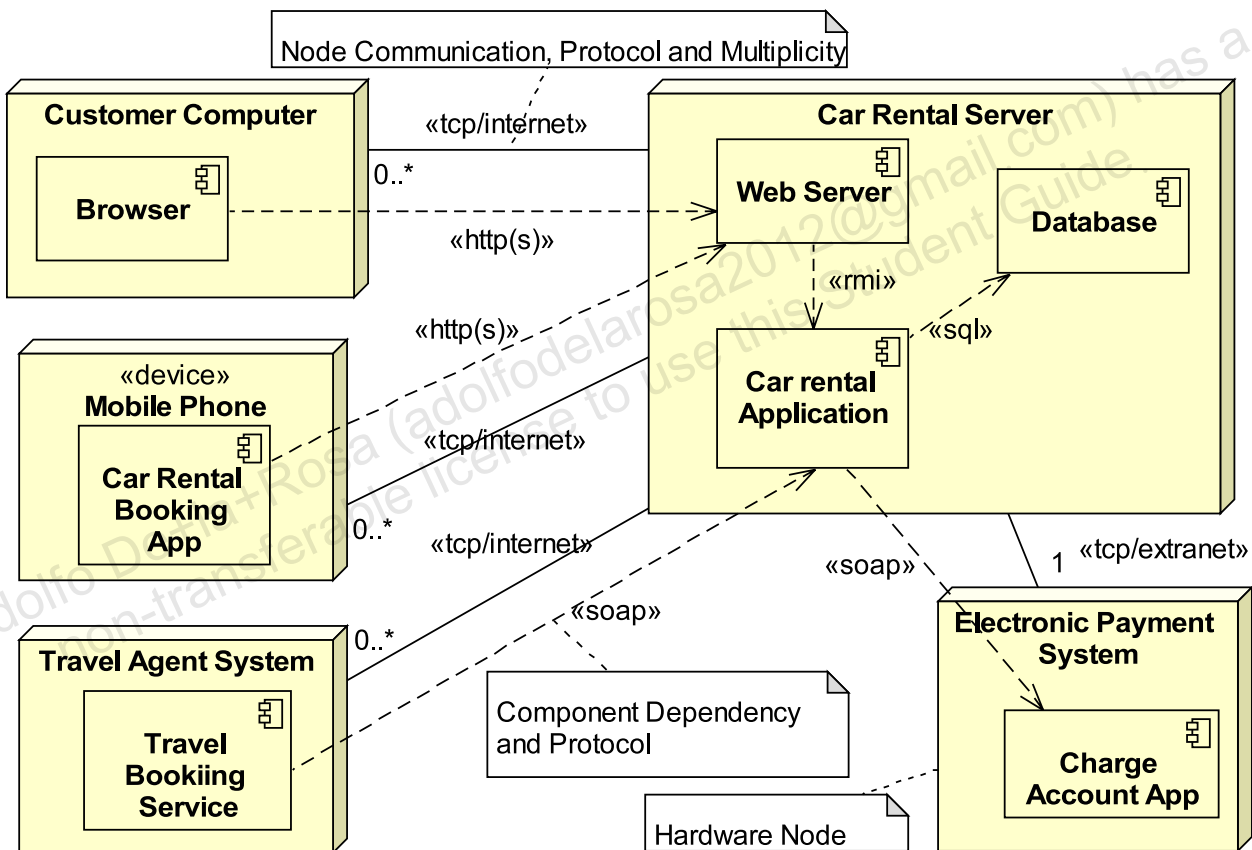


Figure 11-20 An Example Deployment Diagram

### 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

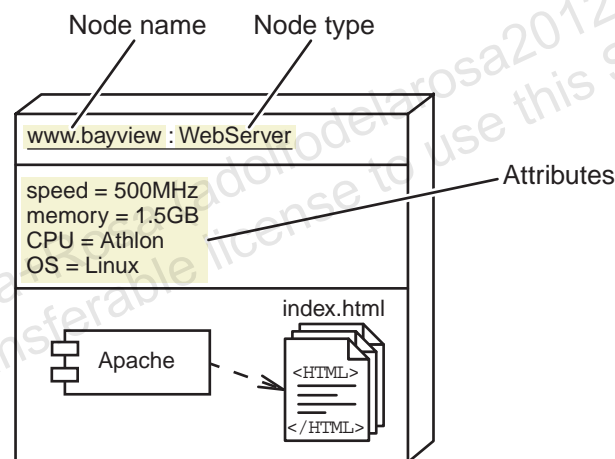
Deployment diagrams come in two forms:

- 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.

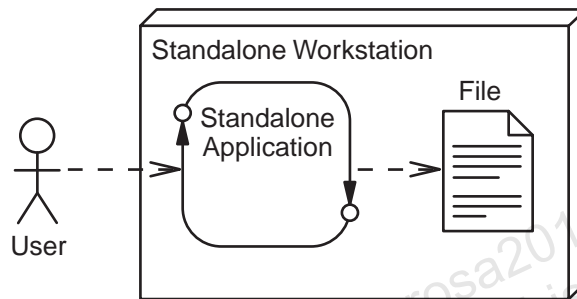
There are hundreds of successful software architectures. A few common types are:

- 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.



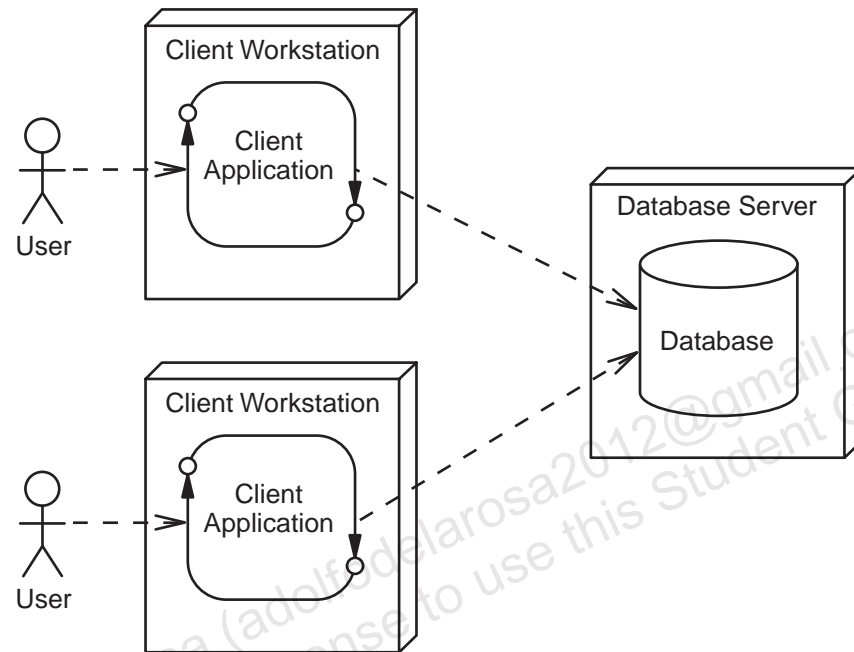
**Figure 11-22** Generic Standalone Architecture Type

Standalone applications are very common. Examples include word processors, graphics applications, text editors, and so on. These types of applications do 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.

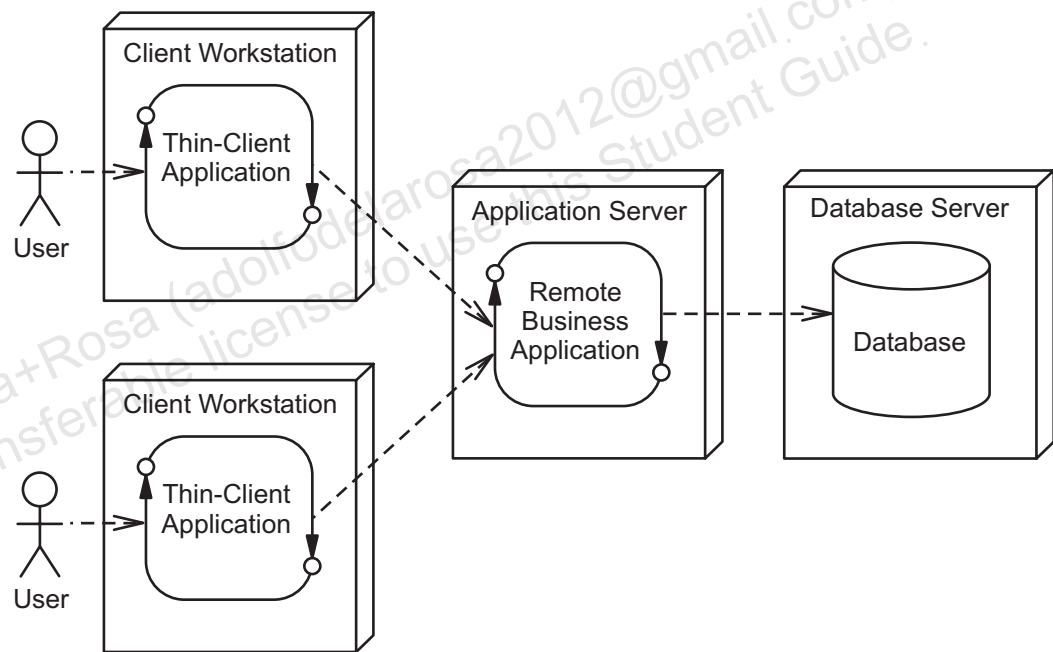
This architecture type also suffers from the deployment problem. This problem is significant in this situation, because every client must have the latest version of the client application when the data store is upgraded. Every user must be using the same version of the software or data corruption problems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

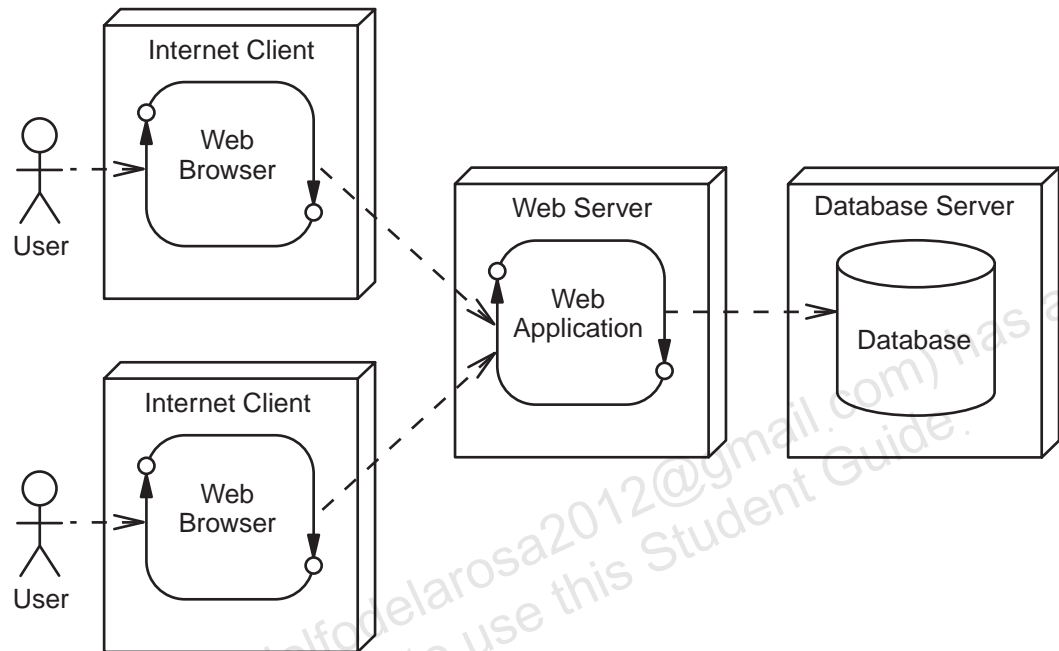


**Figure 11-24** Generic N-tier Architecture Type

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 be 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.

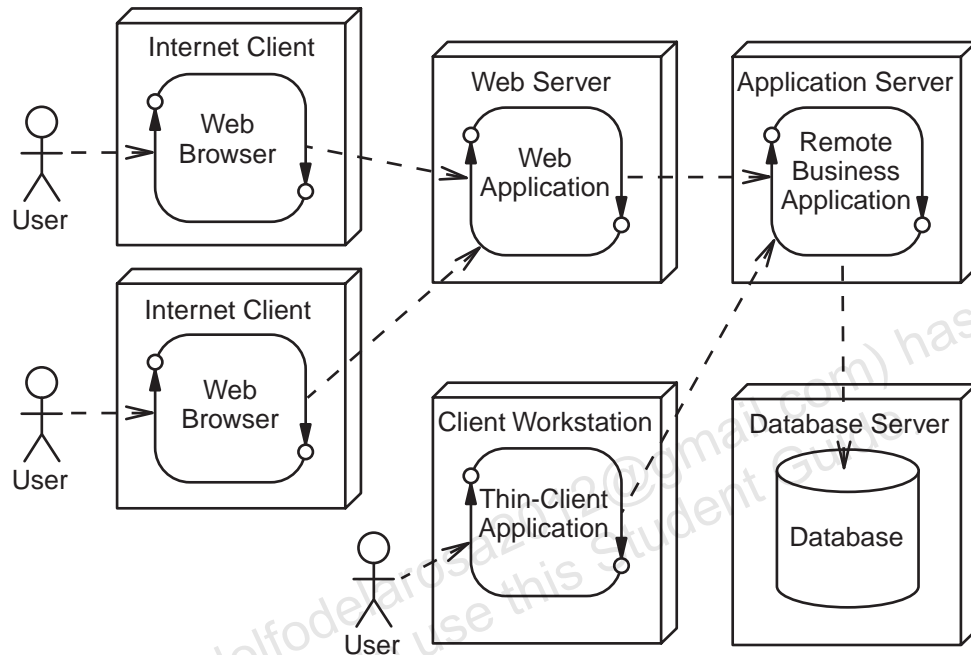


**Figure 11-25** Web-Centric N-Tier Architecture Type

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.



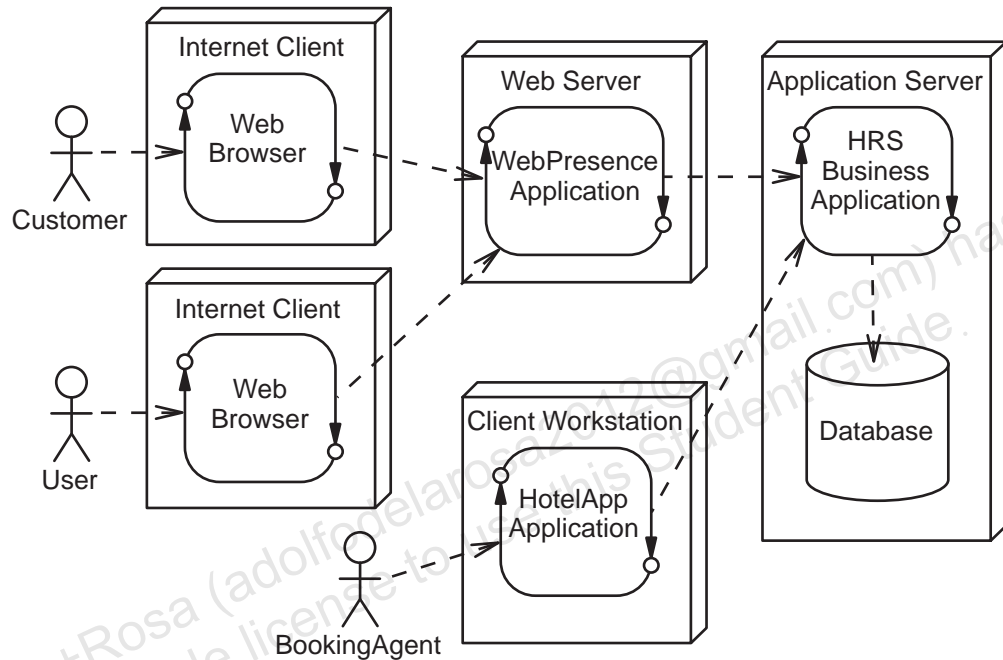
**Figure 11-26** Enterprise N-Tier Architecture Type

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

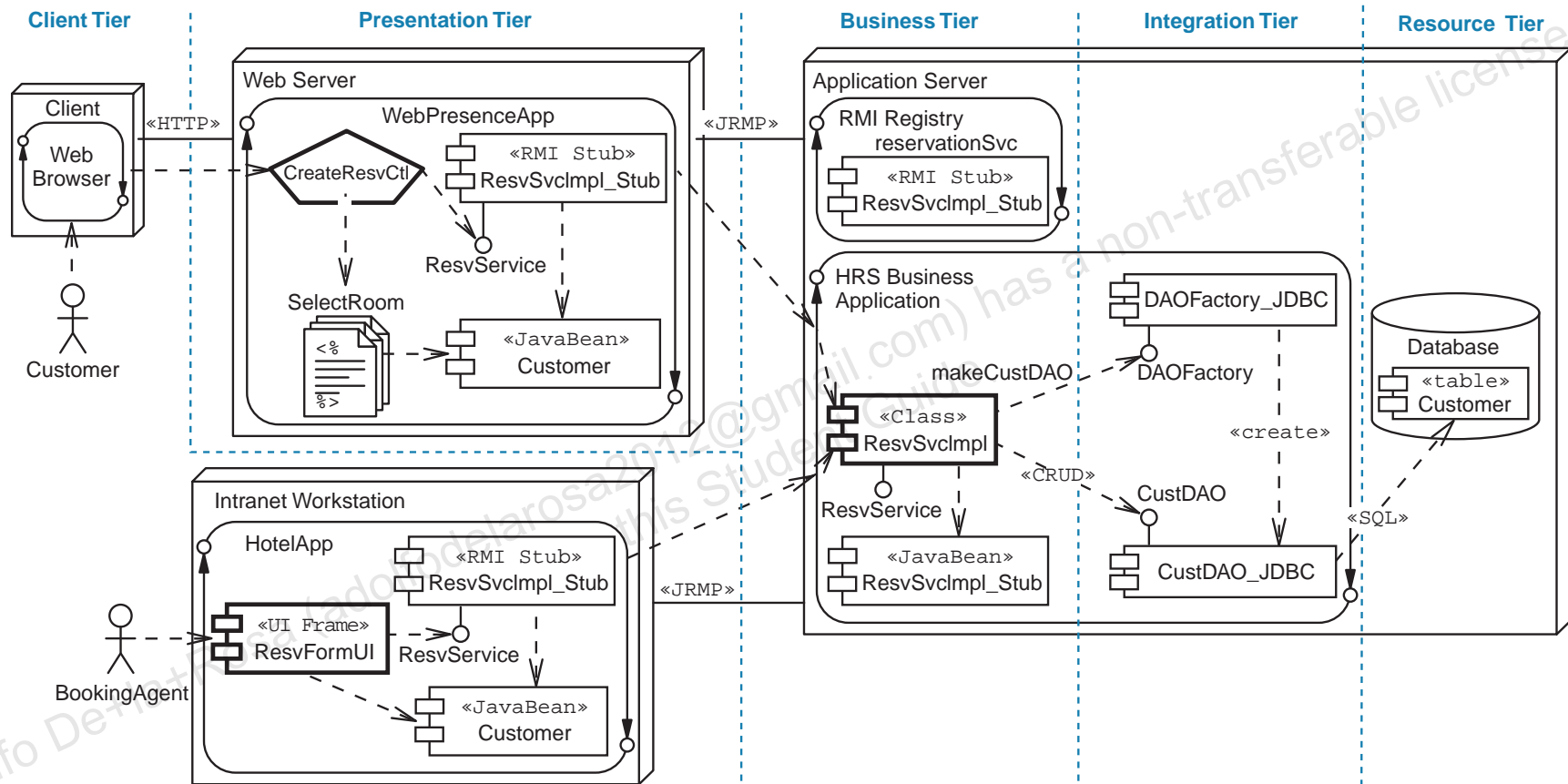
This section describes how to build a Detailed Deployment diagram. This is an essential part of the Architecture model because it shows the actual components that need to be developed. The Detailed Deployment diagram also acts as a basis for the Architecture template which the development team will use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

## Creating the Architecture Workflow Artifacts



**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.

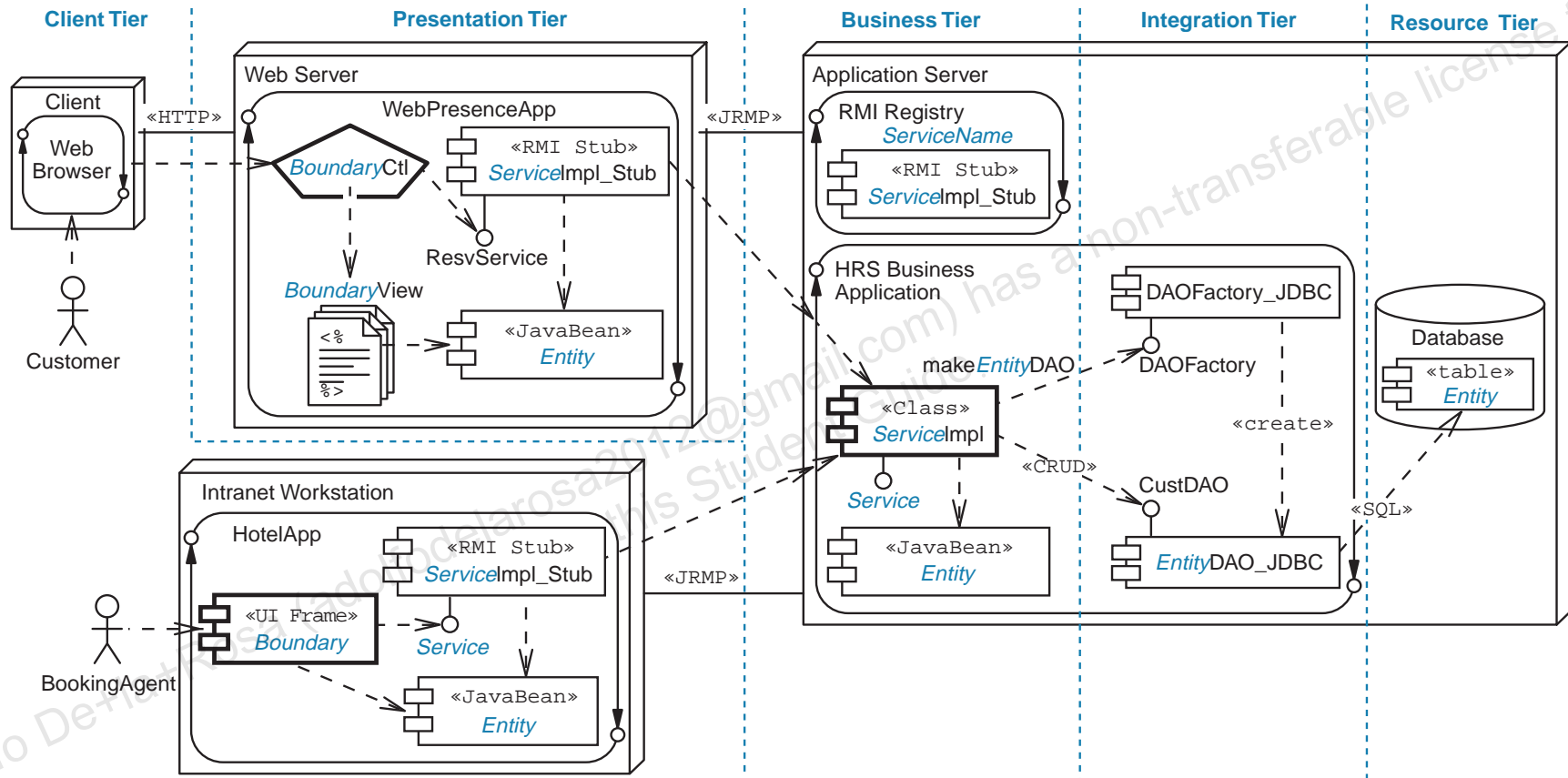
The idea is to show the types of infrastructure components needed to support the Design components. You can simplify the detailed Deployment diagram by eliminating all redundant components. For example, only include one Entity type component (in each tier).

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.

## Creating the Architecture Workflow Artifacts



**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.

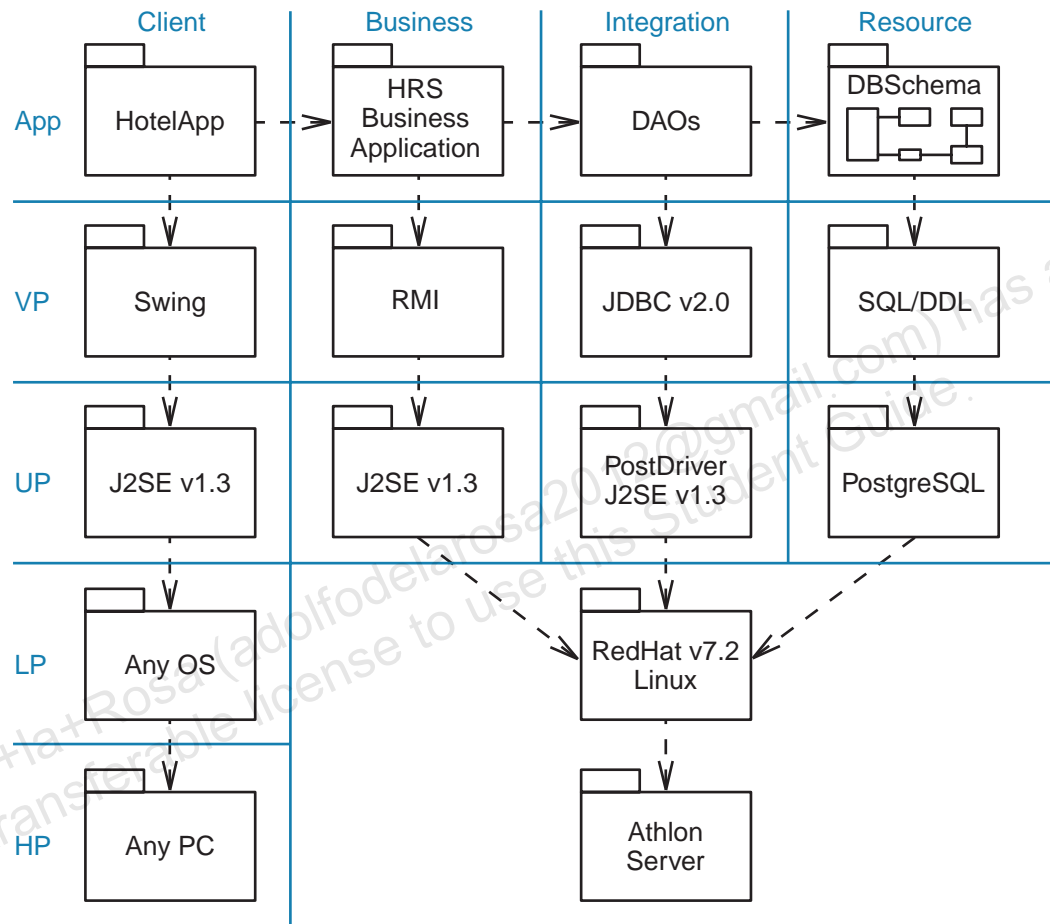
4. Determine which operating system to use.

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

5. Determine what hardware to use.

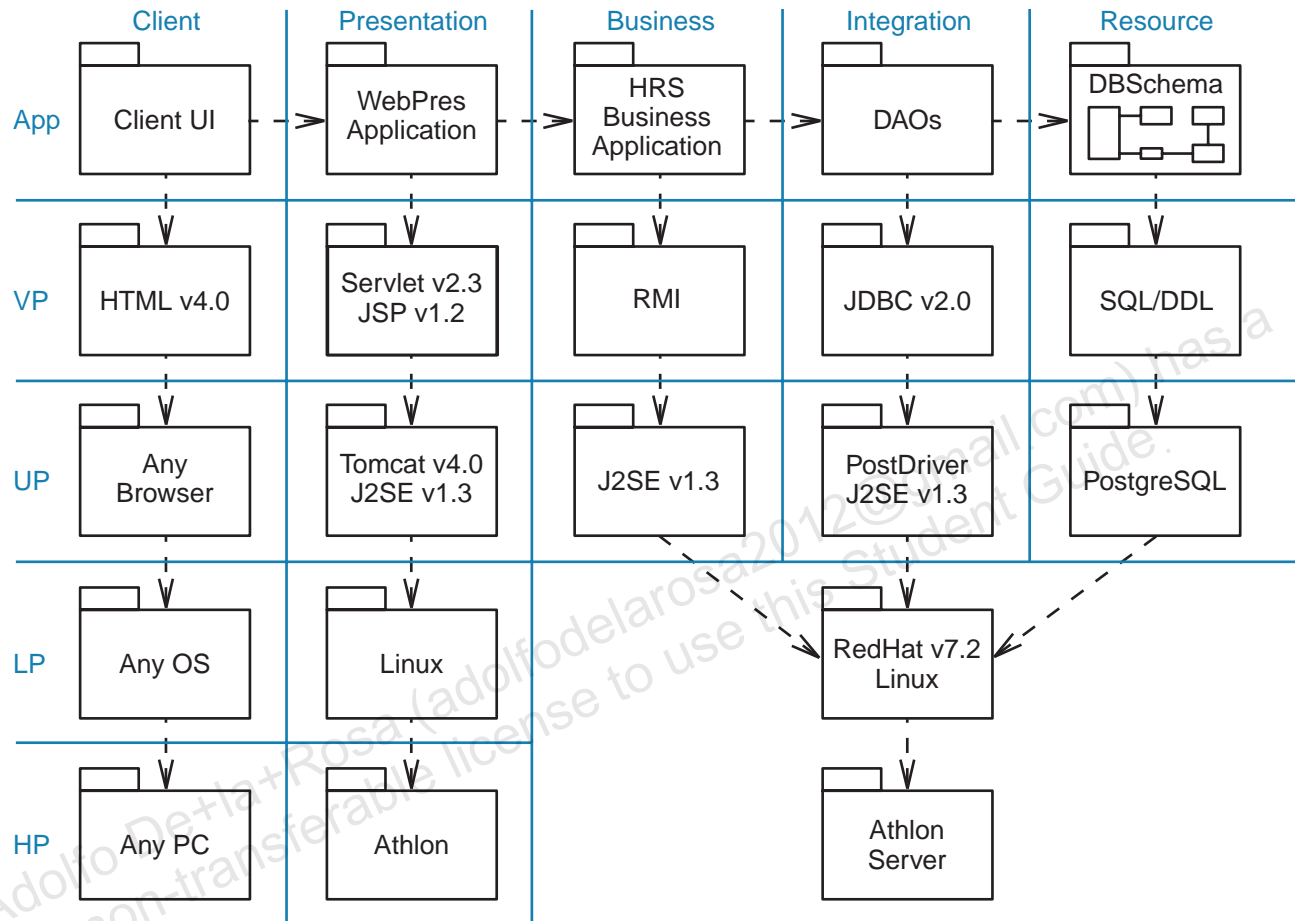
In the Hardware Platform layer, document the type of machine that will host the given tier's components. Be as specific as you can with details about the number of processors, processor speed, memory capacity, and storage capacity.

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