Object-Oriented Analysis and Design Using UML

Volume II • Student Guide

OO-226 Rev E

D61808GC21

Edition 2.1

June 2010

D67902

ORACLE®

This page intentionally left blank.

This page intentionally left blank.

Table of Contents

About This Course ....Preface-xxv
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-xxxi
Conventions .... Preface-xxxii
Icons .... Preface-xxxii
Typographical Conventions .... Preface-xxxii
Additional Conventions .... Preface-xxxiii

Examining Object-Oriented Concepts and Terminology 1-1
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

Introducing Modeling and the Software Development Process 2-1
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

Creating Use Case Diagrams 3-1
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

Creating Use Case Scenarios and Forms 4-1
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

Creating Activity Diagrams..5-1
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

Determining the Key Abstractions...6-1
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

Constructing the Problem Domain Model....................................7-1
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 Using Interaction Diagrams 8-1
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

Modeling Object State Using State Machine Diagrams 9-1
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

Applying Design Patterns to the Design Model 10-1
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

Introducing Architectural Concepts and Diagrams 11-1
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

Introducing the Architectural Tiers 12-1
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

Refining the Class Design Model 13-1
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

Overview of Software Development Processes 14-1
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

Overview of Frameworks 15-1
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

Course Review 16-1
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

Drafting the Development Plan A-1
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

Constructing the Software Solution B-1
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

Testing the Software Solution C-1
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

Deploying the Software Solution D-1
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

Quick Reference for UML E-1
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

Additional Resources... F-1
Additional Resources F-1

Glossary/Acronyms... Glossary-1

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 `ReservationServic`e 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

About This Course

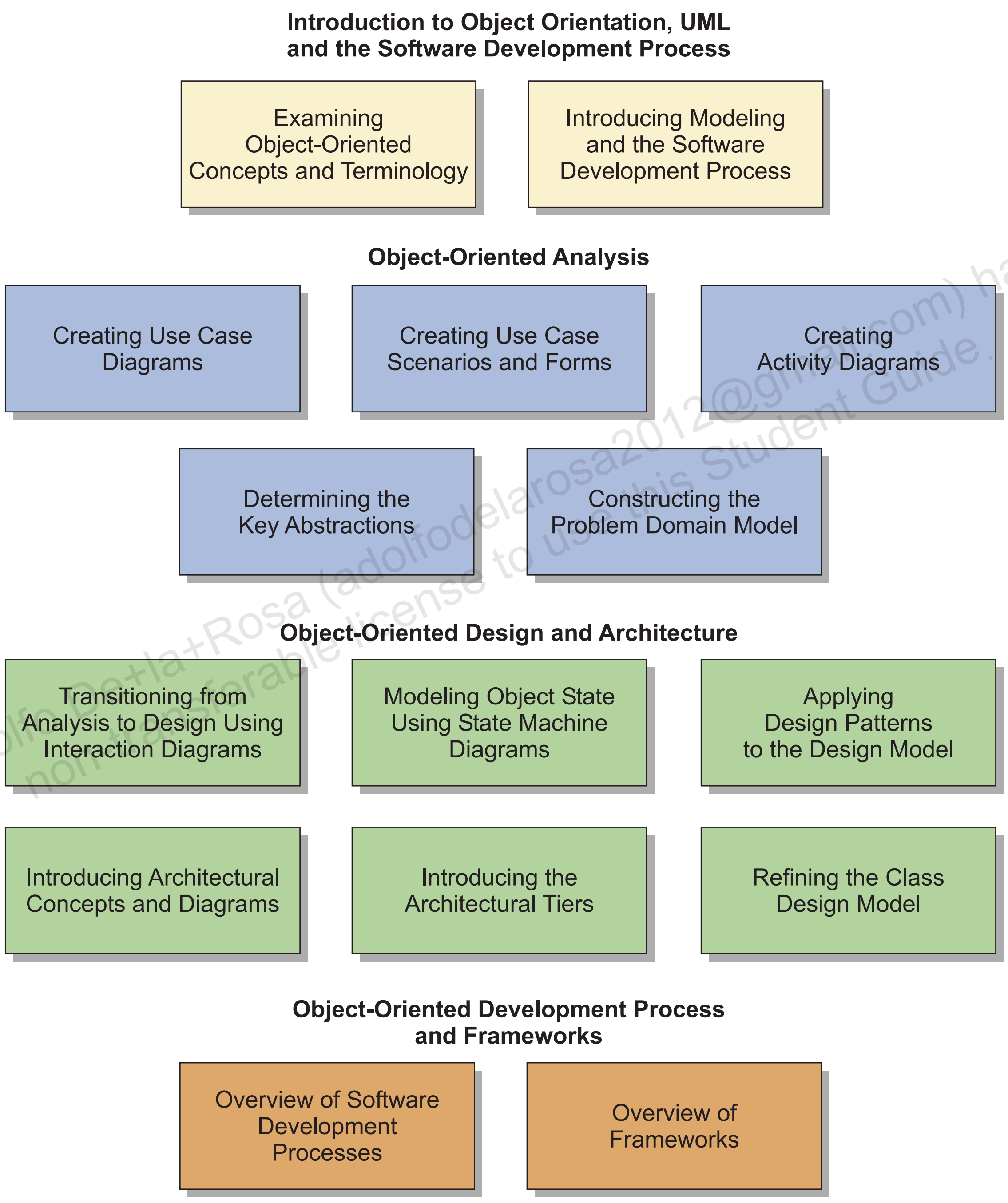
Course Goals

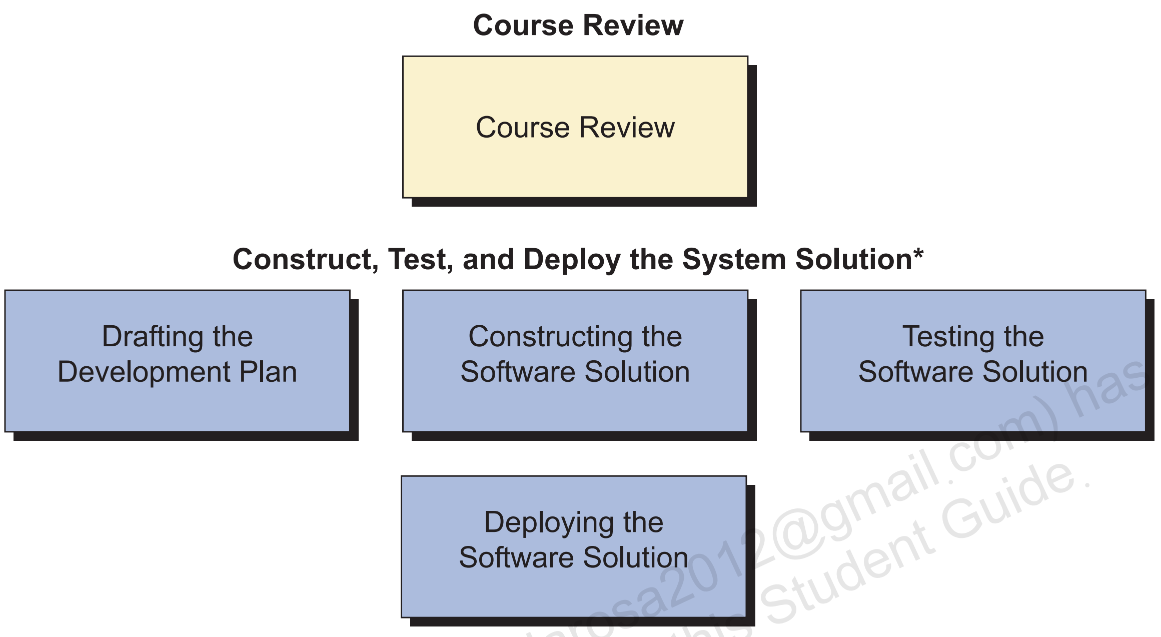
Upon completion of this course, you should be able to:

- 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+rwx *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
  ```

Module 12

Introducing the Architectural Tiers

Objectives

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

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

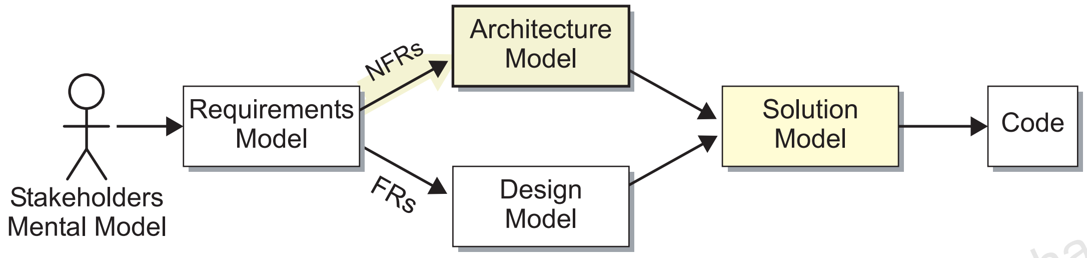
Additional Resources

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

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

Process Map

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



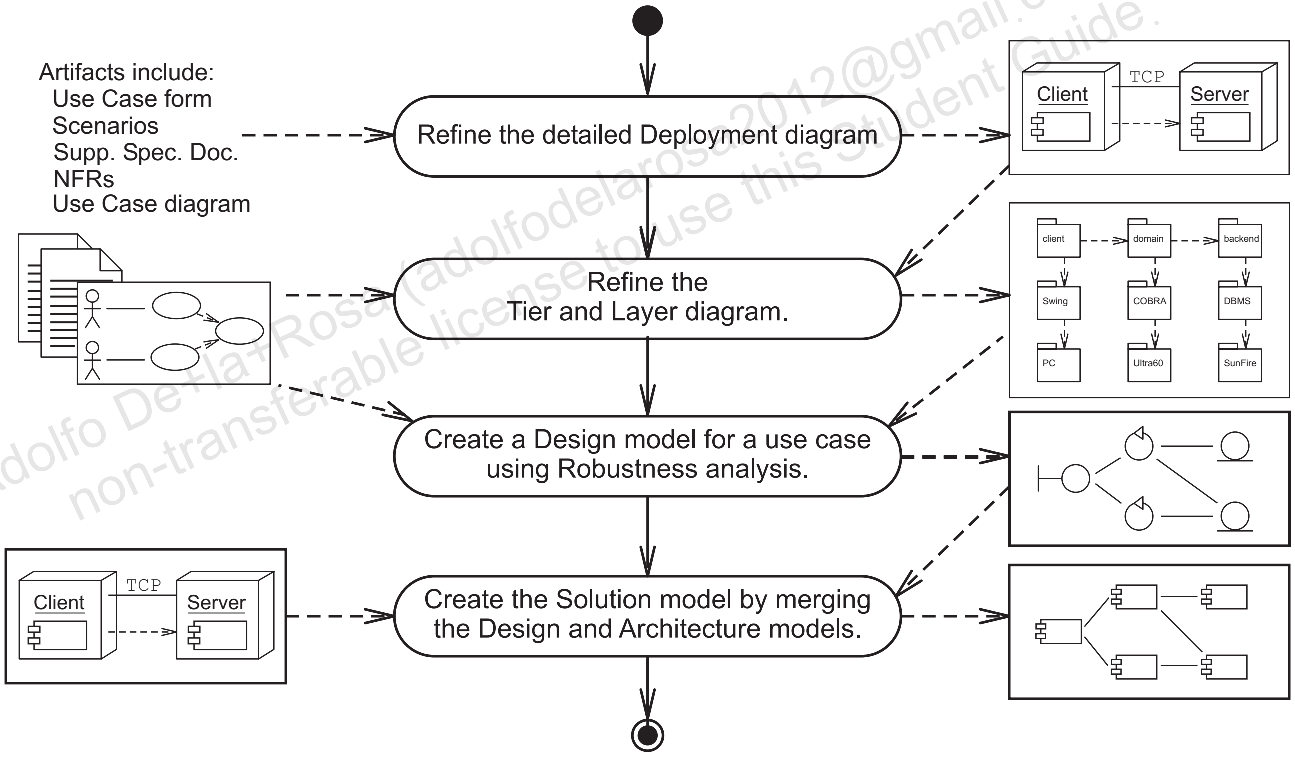

Artifacts include:
Use Case form
Scenarios
Supp. Spec. Doc.
NFRs
Use Case diagram
Refine the detailed Deployment diagram
Refine the Tier and Layer diagram.
Create a Design model for a use case using Robustness analysis.
Create the Solution model by merging the Design and Architecture models.
Client
Server
TCP
client
domain
backend
Swing
COBRA
DBMS
PC
Ultra60
SunFire

Figure 12-1 Architecture Tiers Process Map

Introducing the Client and Presentation Tiers

The Client and Presentation tiers primarily contain:

- Controller components:
  - Control the input from the boundary
  - Perform input sanity checking

    Input sanity checking will ensure that the data submitted is plausible. For example, it can check that the dates entered are valid.
  - Call business logic methods
  - Dispatch view components
- View components:
  - Retrieve data required by the view
  - Prepare the view in a format suitable for the recipient
  - Add client-side sanity checking (optional)

    Client-side sanity checking in a Web client could be performed by the browser by using JavaScript™.

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

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

Boundary Interface Technologies

The primary Boundary Interface types are:

- Graphical user interface (GUI)

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

- Web user interface (Web UI)

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

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

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

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

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

There are four fundamental types of application components:

- Controller

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

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

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

These components represent business entities (domain objects).

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

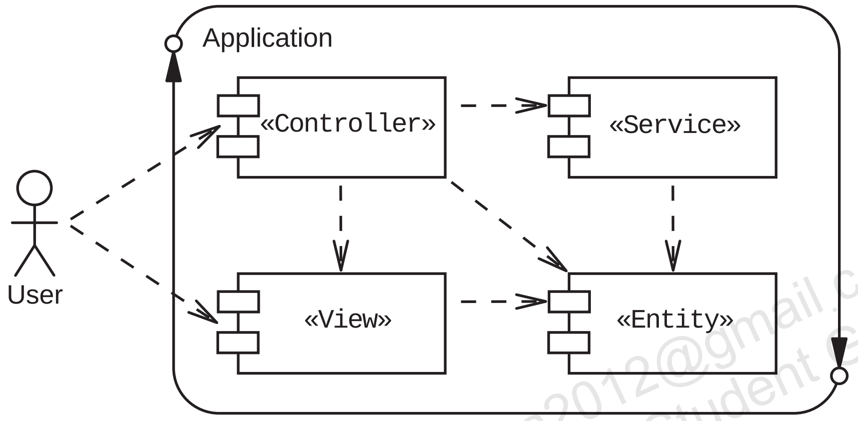


Figure 12-2 Generic Application Components

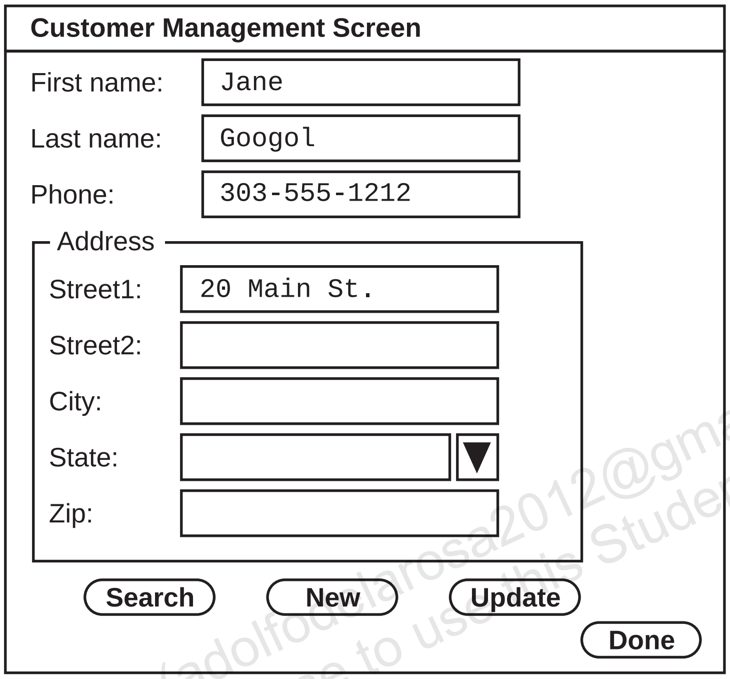
This module introduces you to the technologies that create controller and view components for GUI and Web UI systems.

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

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

GUI Screen Design

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



Customer Management Screen

First name: Jane

Last name: Googol

Phone: 303-555-1212

Address

Street1: 20 Main St.

Street2:

City:

State:

Zip:

Figure 12-3 Customer Management Screen

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

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

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

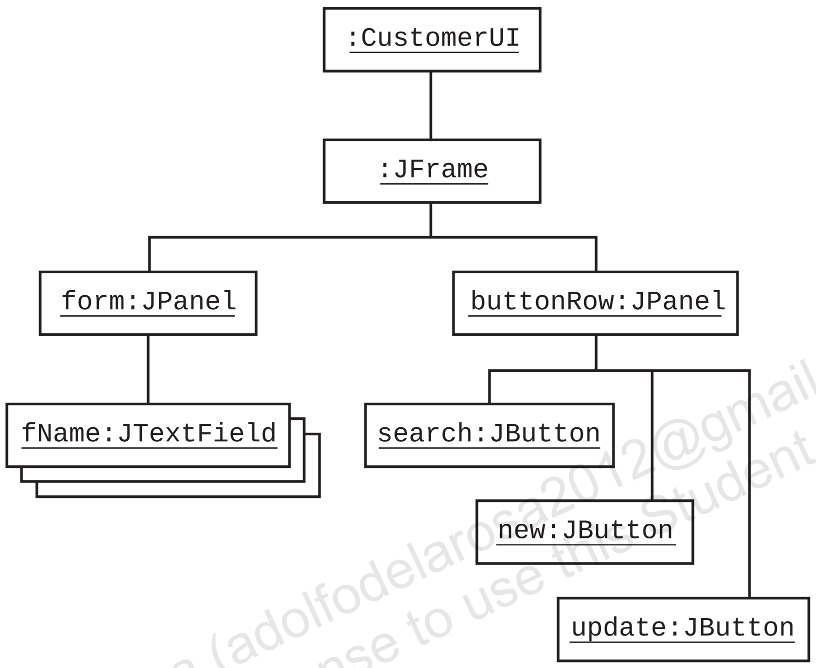


Figure 12-4 CustomerUI GUI Component Hierarchy

GUI Event Model

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

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

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

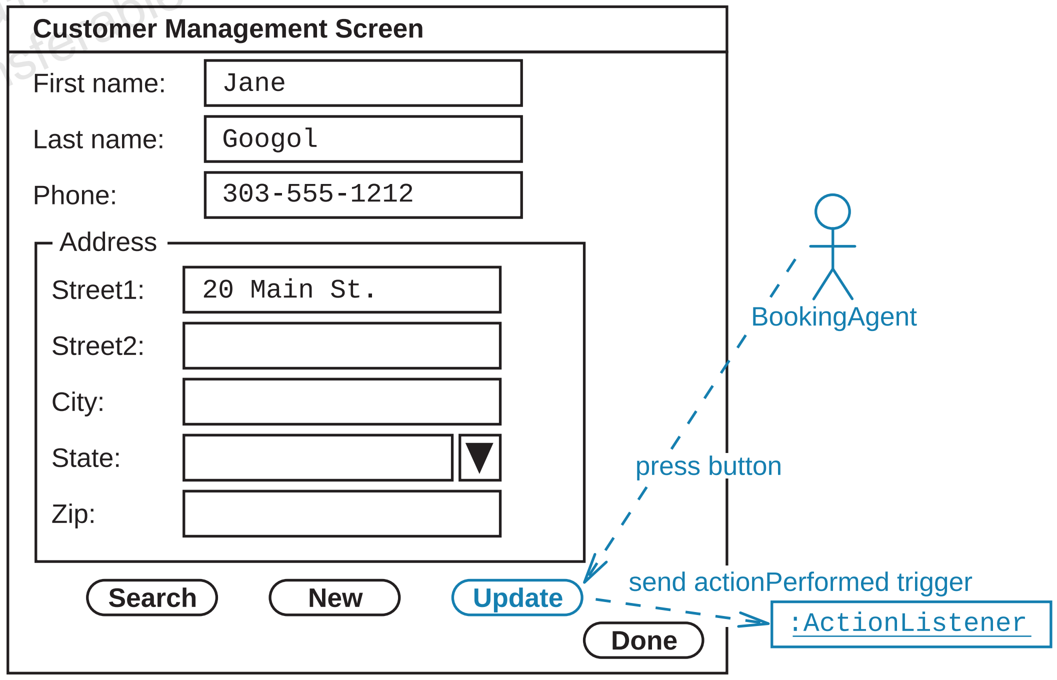


Figure 12-5 Java Technology Event Model

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

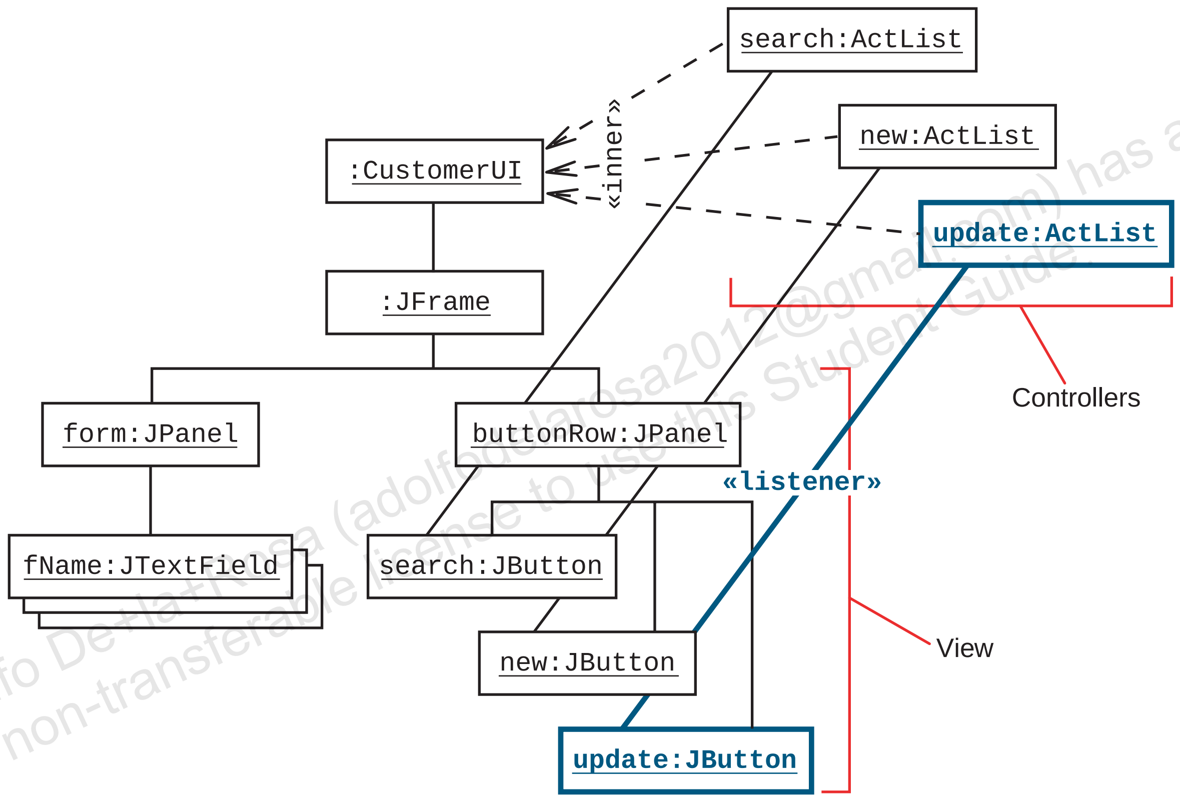


Figure 12-6 GUI Listeners as Controller Elements

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

MVC Pattern

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

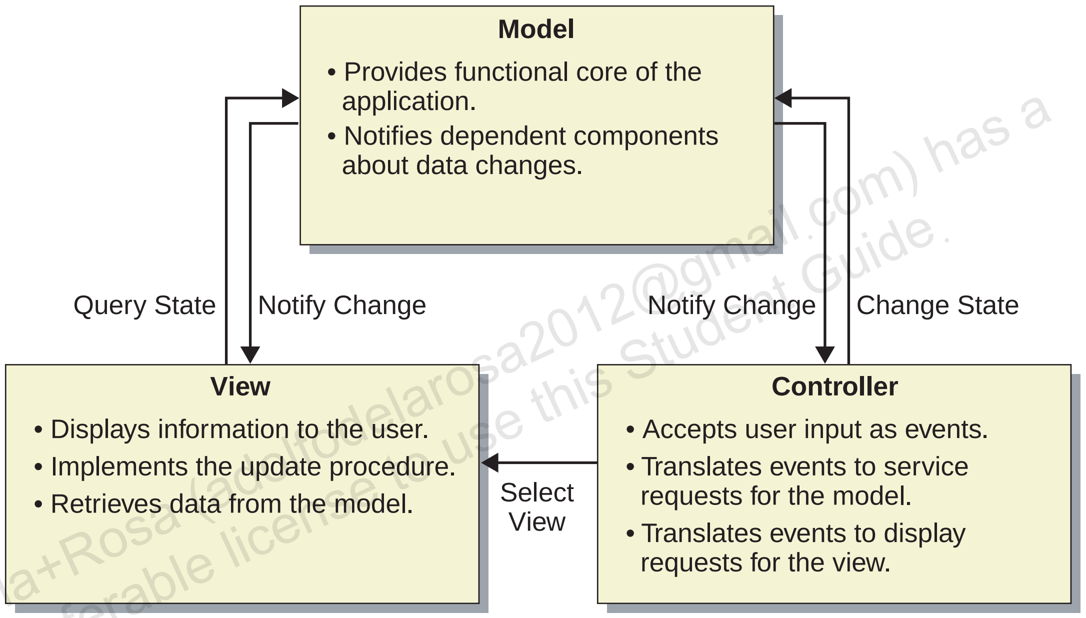

Model
• Provides functional core of the application.
• Notifies dependent components about data changes.
Query State
Notify Change
Notify Change
Change State
View
• Displays information to the user.
• Implements the update procedure.
• Retrieves data from the model.
Select View
Controller
• Accepts user input as events.
• Translates events to service requests for the model.
• Translates events to display requests for the view.

Figure 12-7 MVC Pattern

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

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

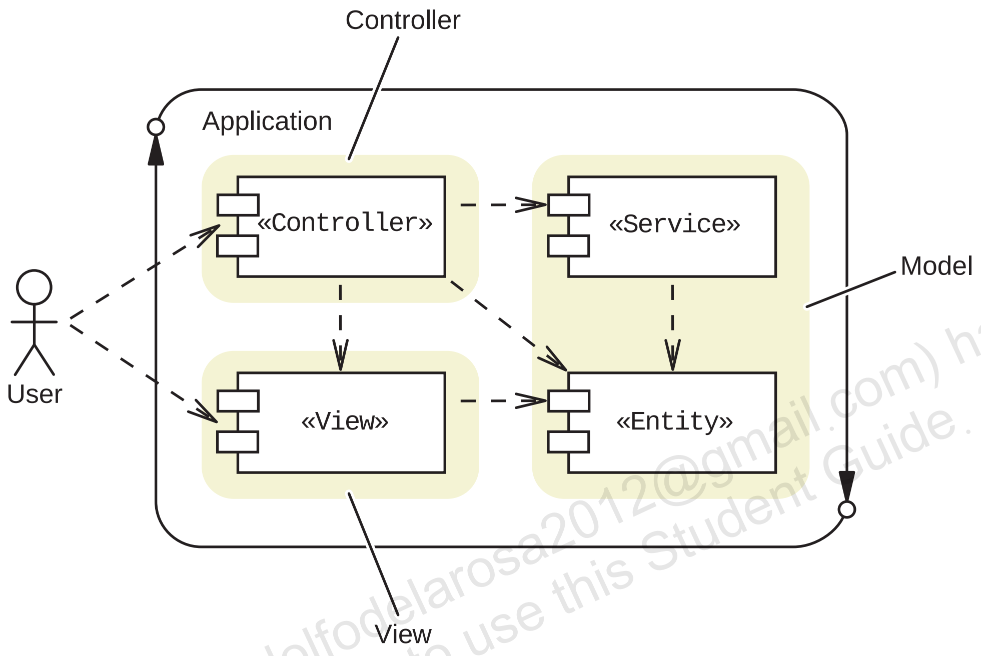


Figure 12-8 MVC Component Types

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

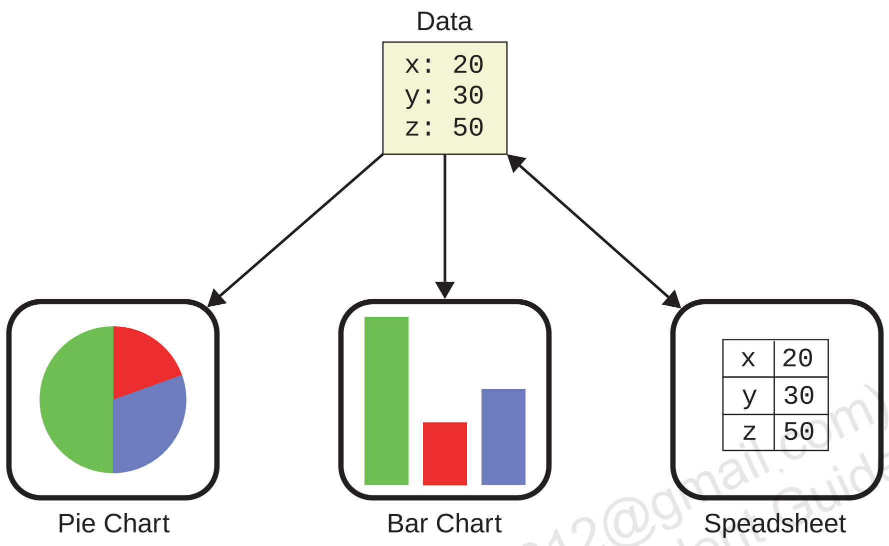


Figure 12-9 Example Use of the MVC Pattern

Overview of the Tiers and Layers Package Diagram

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

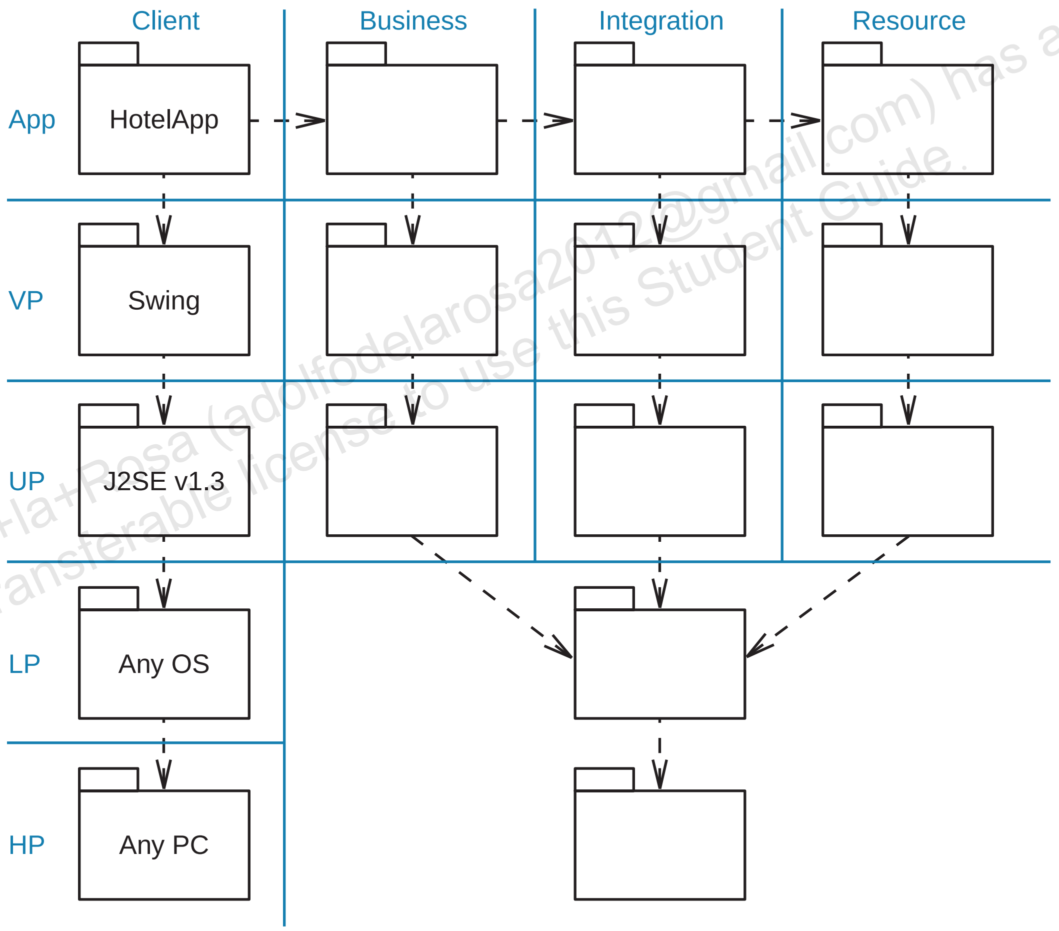


Figure 12-10 A Partial Tiers and Layers Package Diagram for the HotelApp

---

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

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

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

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

Web UIs have the following characteristics:

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

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

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

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

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

- There is often a single path through the screens.

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

- Only one screen is usually open at a time.

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

Web UI Screen Design

The following considerations affect the design of Web UI screens:

- A Web UI tends to be constructed as a sequence of related screens.

  The Web UI page sequence usually implements the workflow of a single use case.
- Each screen is a hierarchy of UI components.

  Each Web UI page can include much information. Typically, a Web UI page will include an HTML form that tells the Web browser to create corresponding GUI components, such as text fields.
- A Web UI screen presents the user's view of the domain model as well as presents the user's action controls.

  A Web UI page presents the state of the application using HTML features, such as tables. A Web UI page can use HTML forms to permit users to interact with the application.
- It is rare that a screen can be reused by multiple use cases.

  Because Web UI screens tend to be designed specifically for a single use case, this usually prevents these components from being reused in other use case workflows. There are a few notable exceptions. For example, login or *create user* screens might be reused in a variety of use cases.

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

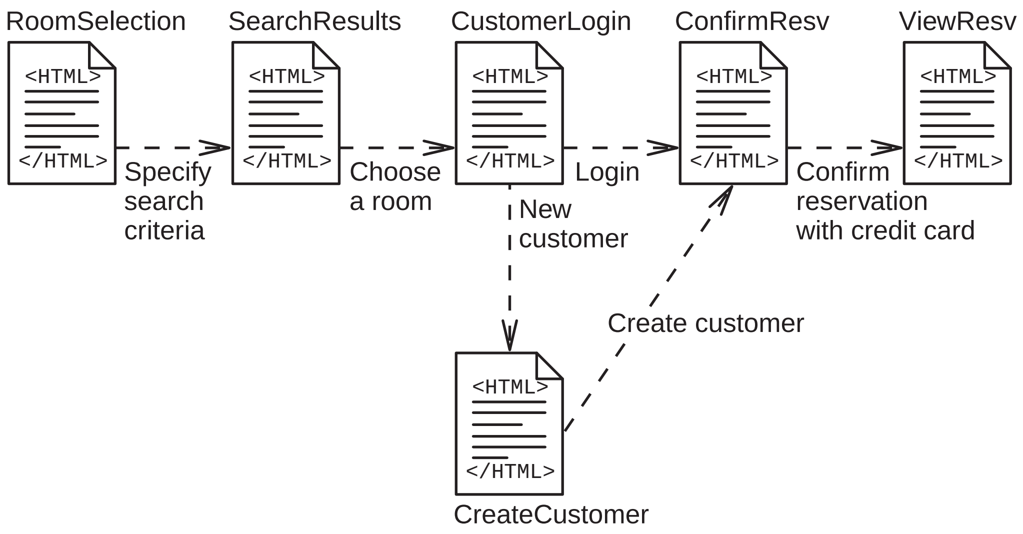


Figure 12-11 Example Web Page Flow

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

Code 12-1 Partial Web UI Form Example

```
<FORM ACTION='makeResv' METHOD='POST'>
  <INPUT TYPE='hidden' NAME='action' VALUE='roomSearch'>
  Enter arrival date: <INPUT TYPE='text' NAME='arrivalDate'>
  <BR>
  Enter departure date: <INPUT TYPE='text' NAME='departureDate'>
  <BR>
  Select room type:
  <SELECT NAME='roomType'>
    <OPTION VALUE='Single'> Single
    <OPTION VALUE='Double'> Double
    <OPTION VALUE='Suite'> Suite
  </SELECT>
  <BR>
  <INPUT TYPE='submit' VALUE='Search...'>
</FORM>
```

Web UI Event Model

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

- Micro events can be handled by JavaScript™ technology code.

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

- Macro events are handled as HTTP requests from the web browser to the Web server.

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

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

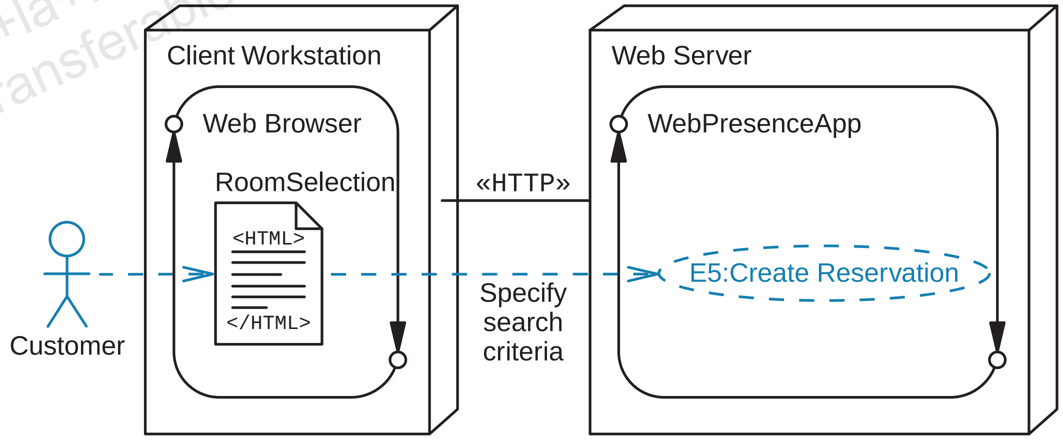


Figure 12-12 Web UI Event Model

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

The WebMVC Pattern

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

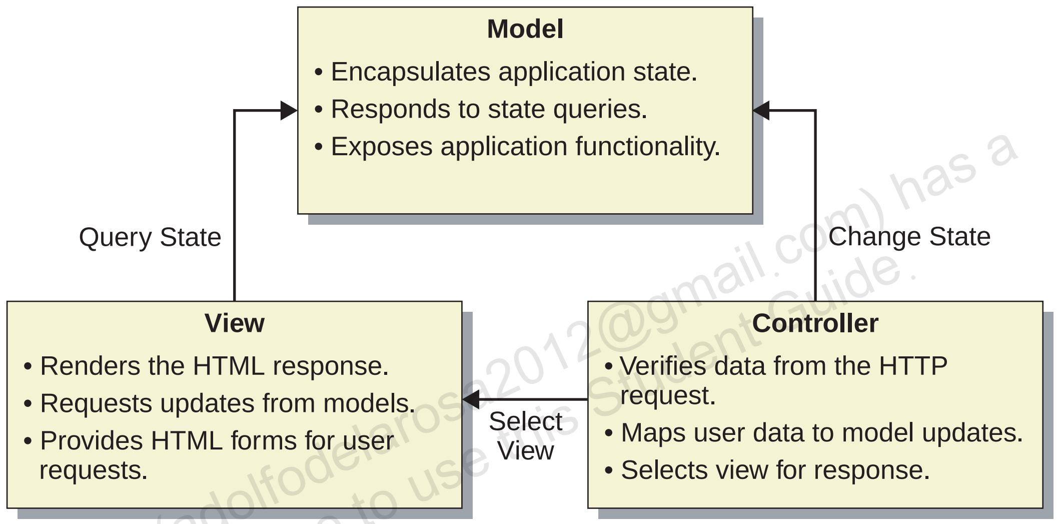

Model
• Encapsulates application state.
• Responds to state queries.
• Exposes application functionality.
Query State
Change State
View
• Renders the HTML response.
• Requests updates from models.
• Provides HTML forms for user requests.
Select View
Controller
• Verifies data from the HTTP request.
• Maps user data to model updates.
• Selects view for response.

Figure 12-13 WebMVC Pattern

The main difference between the GUI MVC and the WebMVC is that there is no mechanism to update the Views when the Model changes[1]. Such functionality would have no meaning for a web application because changed data in the Model cannot be pushed to the Web browser which renders the View. However, even with that limitation, the WebMVC pattern is powerful because of the results of separating these three types of components. For example, it is not uncommon for a Web site to change the look and feel of the site several times a year. By separating the View elements from the Controller elements, the development team can focus on the look-and-feel changes to just the View elements.

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

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

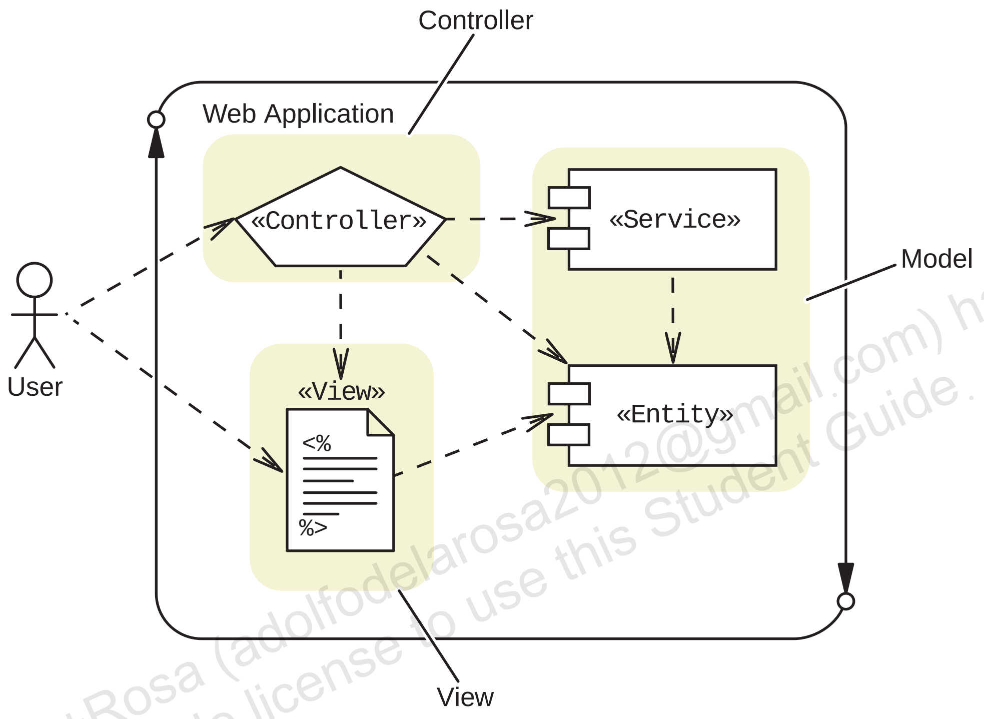


Figure 12-14 WebMVC Pattern Component Types

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

- Java servlets act as a Controller to process HTTP requests:

  User actions in a web application are coarse-grained. A user action is a complete HTTP request, which is usually in response to the user submitting an HTML form. This request contains all of the data on that form. The servlet Controller must perform the following operations:

  - Verify the HTML form data
  - Update the business Model
  - Select and dispatch to the next View (the HTTP response)

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

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

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

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

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

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

Overview of the Tiers and Layers Package Diagram

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

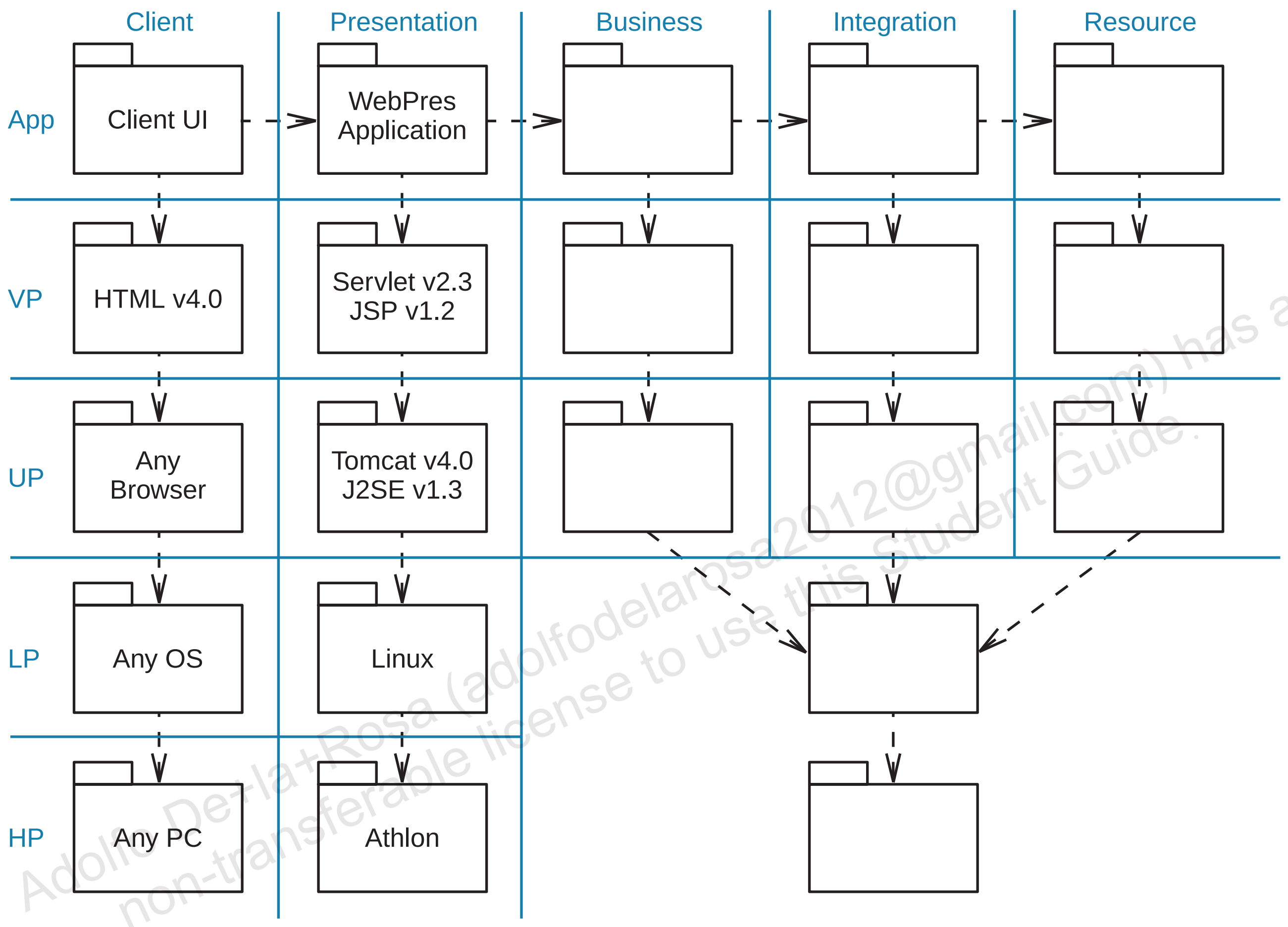


Figure 12-15 A Partial Tiers/Layers Package Diagram

Introducing the Business Tier

The Business tier primarily contains:

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

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

The Business tier may be accessed by:

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

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

Exploring Distributed Object-Oriented Computing

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

- CORBA

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

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

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

- EJB technology

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

- Web services (SOAP)

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

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

Local Access to a Service Component

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

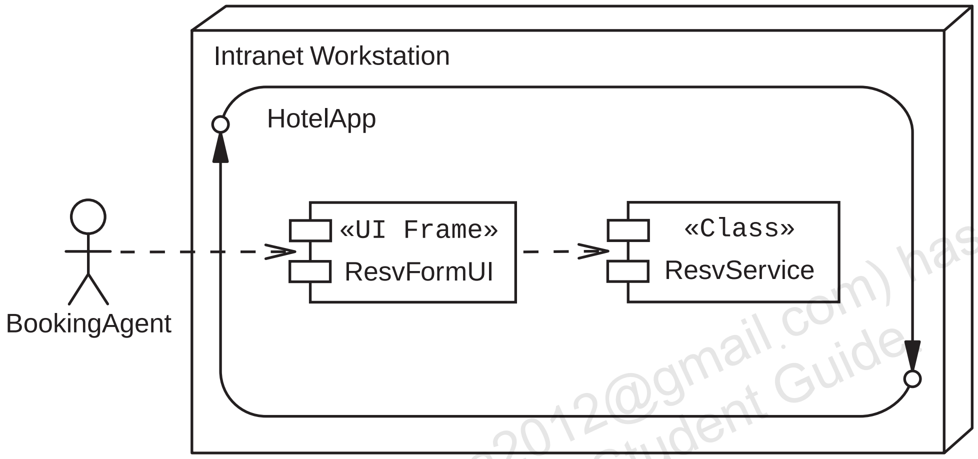


Figure 12-16 Local Access to a Service Component

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

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

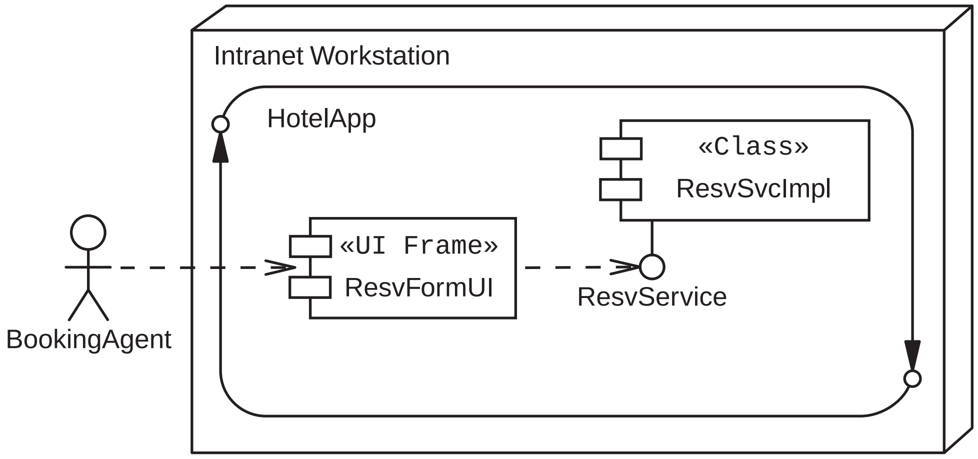


Figure 12-17 Applying the Dependency Inversion Principle

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

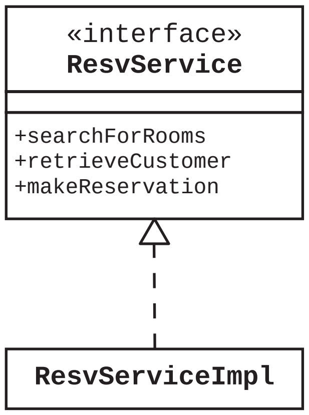


Figure 12-18 Class Diagram of the `ResvService` Interface and Implementation

Remote Access to a Service Component

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

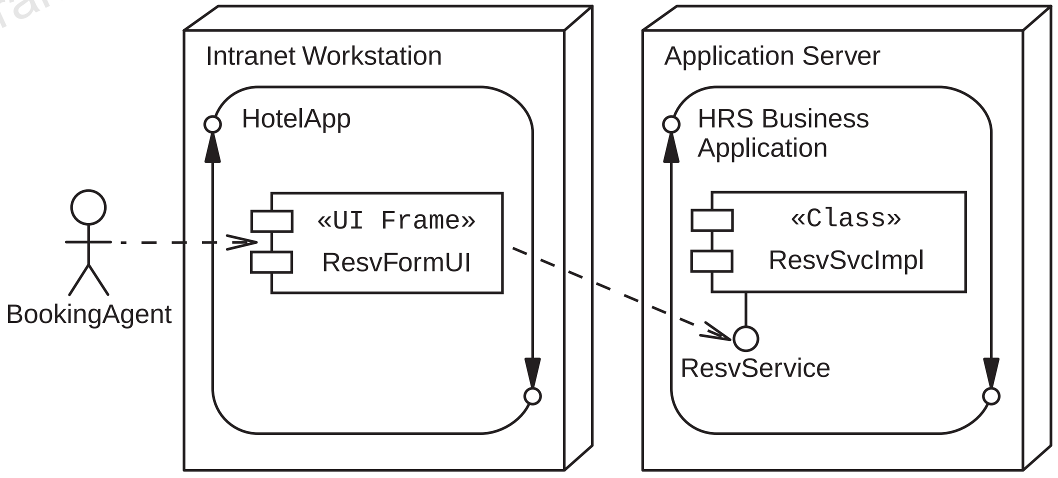


Figure 12-19 An Abstract Version of Accessing a Remote Service

Remote Access Using RMI

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

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

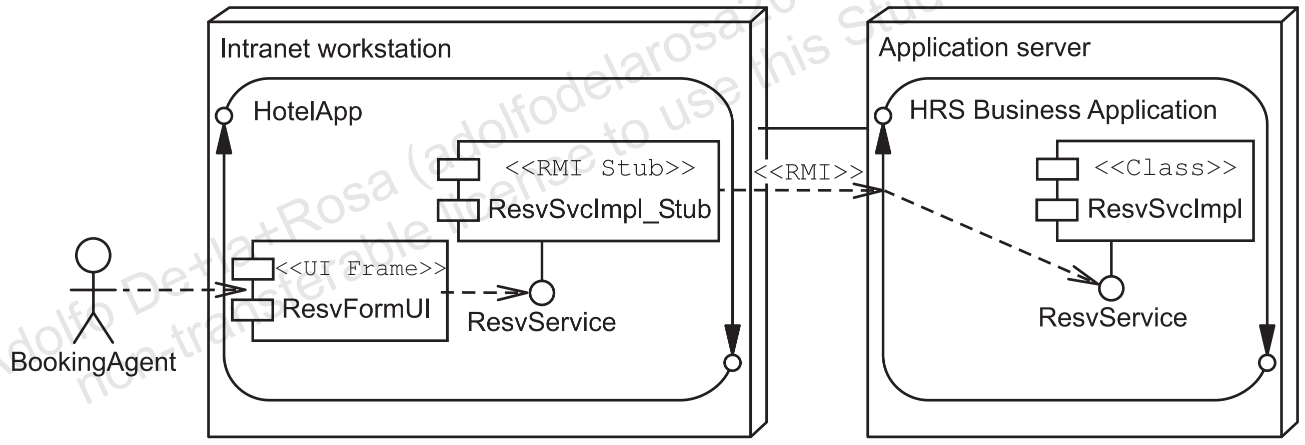


Figure 12-20 Accessing a Remote Service Without a Skeleton Component

Parameter Passing Using RMI

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

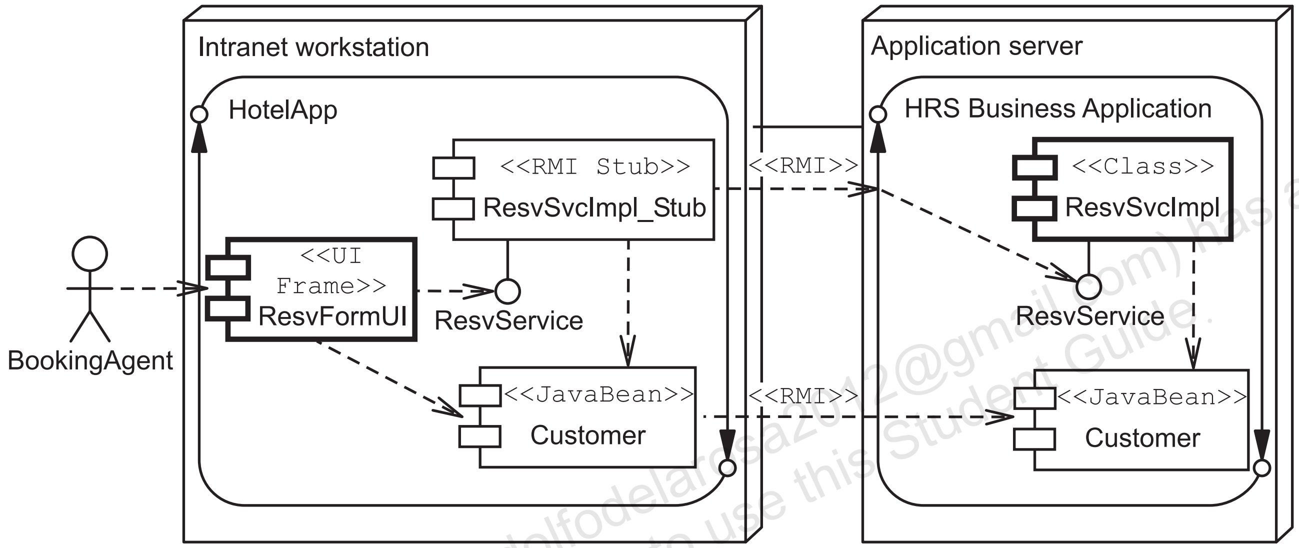


Figure 12-21 RMI Uses Serialization to Pass Parameters

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

Service Lookup Using RMI

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

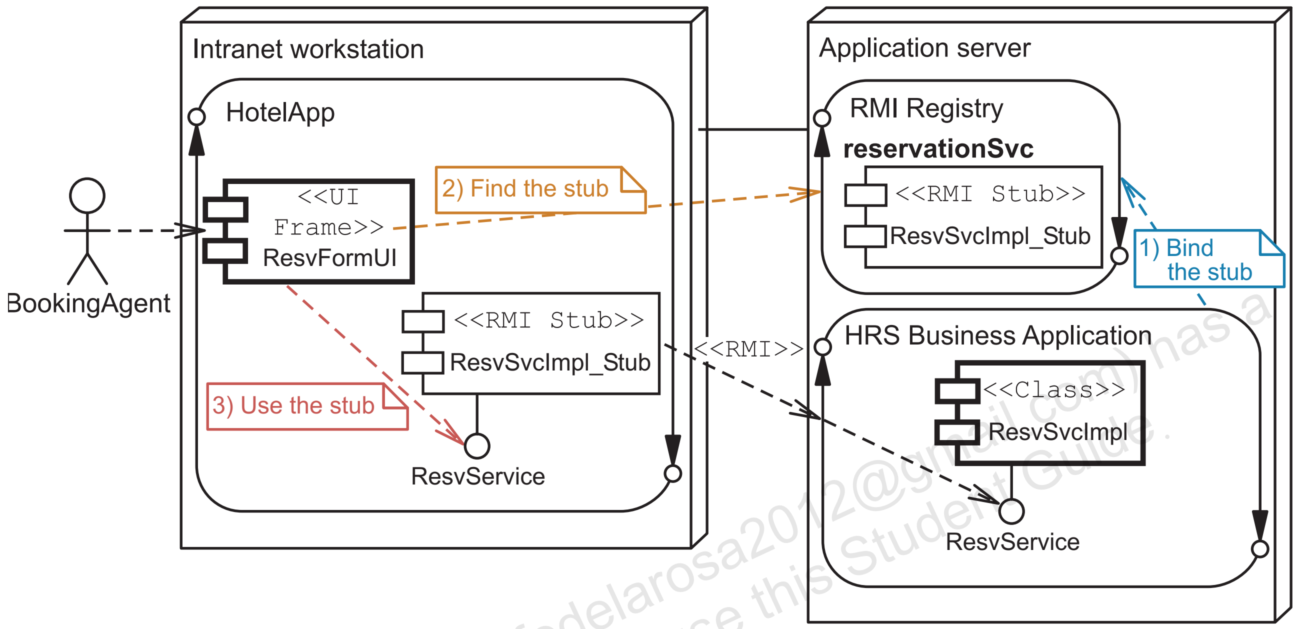


Figure 12-22 RMI Registry Stores Stubs for Remote Lookup

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

Overview of the Detailed Deployment Diagram

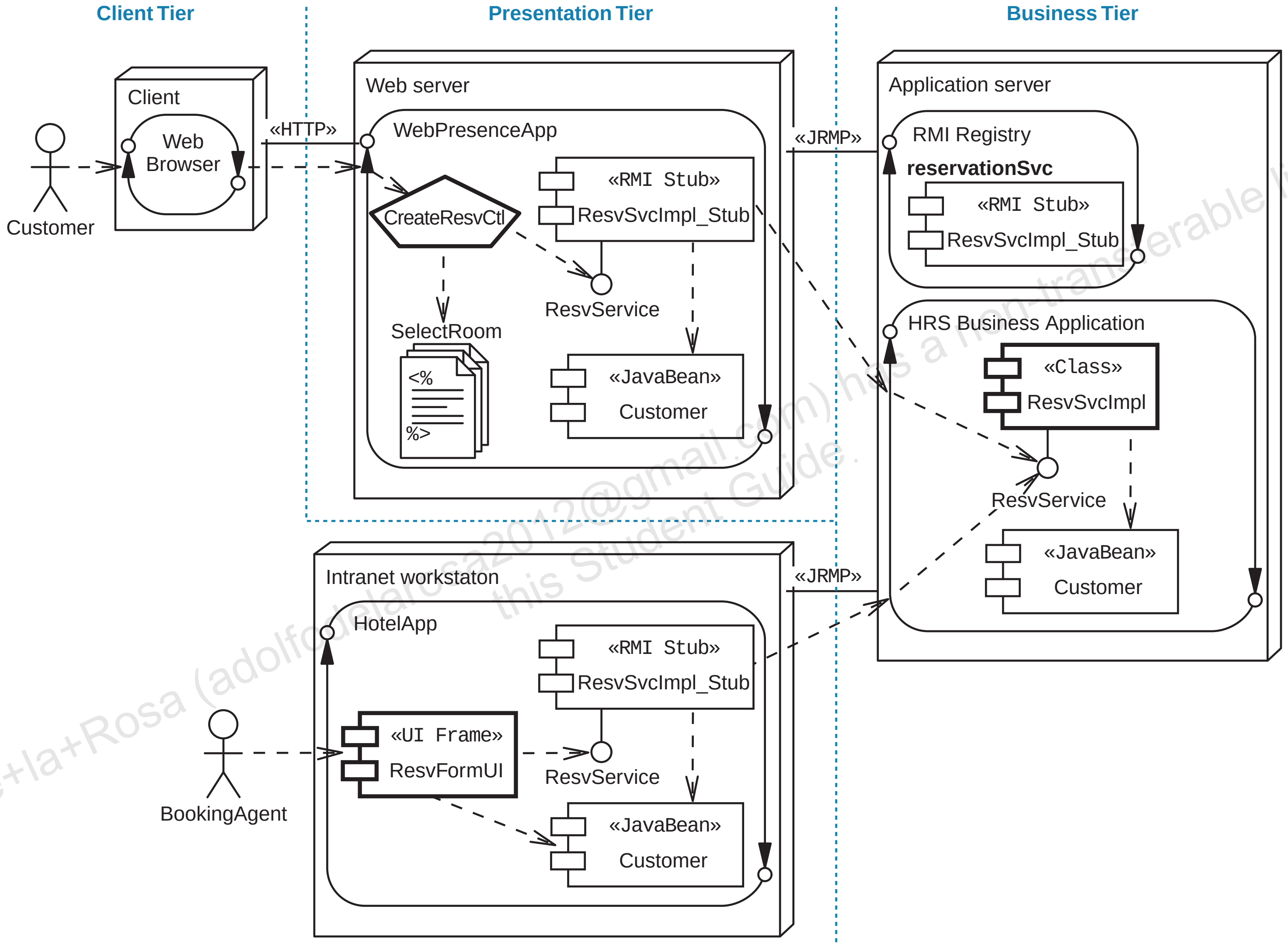


Figure 12-23 Example Detailed Deployment Diagram

Overview of the Tiers and Layers Package Diagram

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

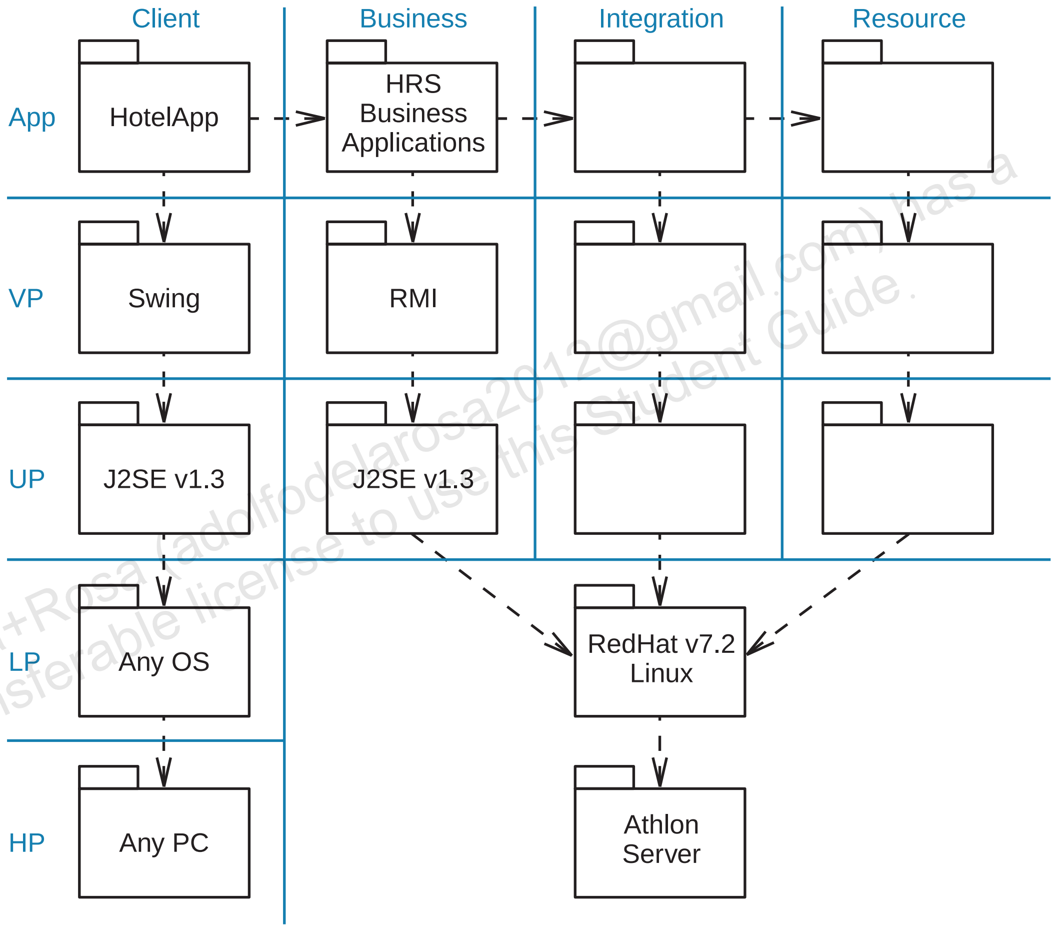


Figure 12-24 Tiers and Layers Diagram for the HotelApp

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

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

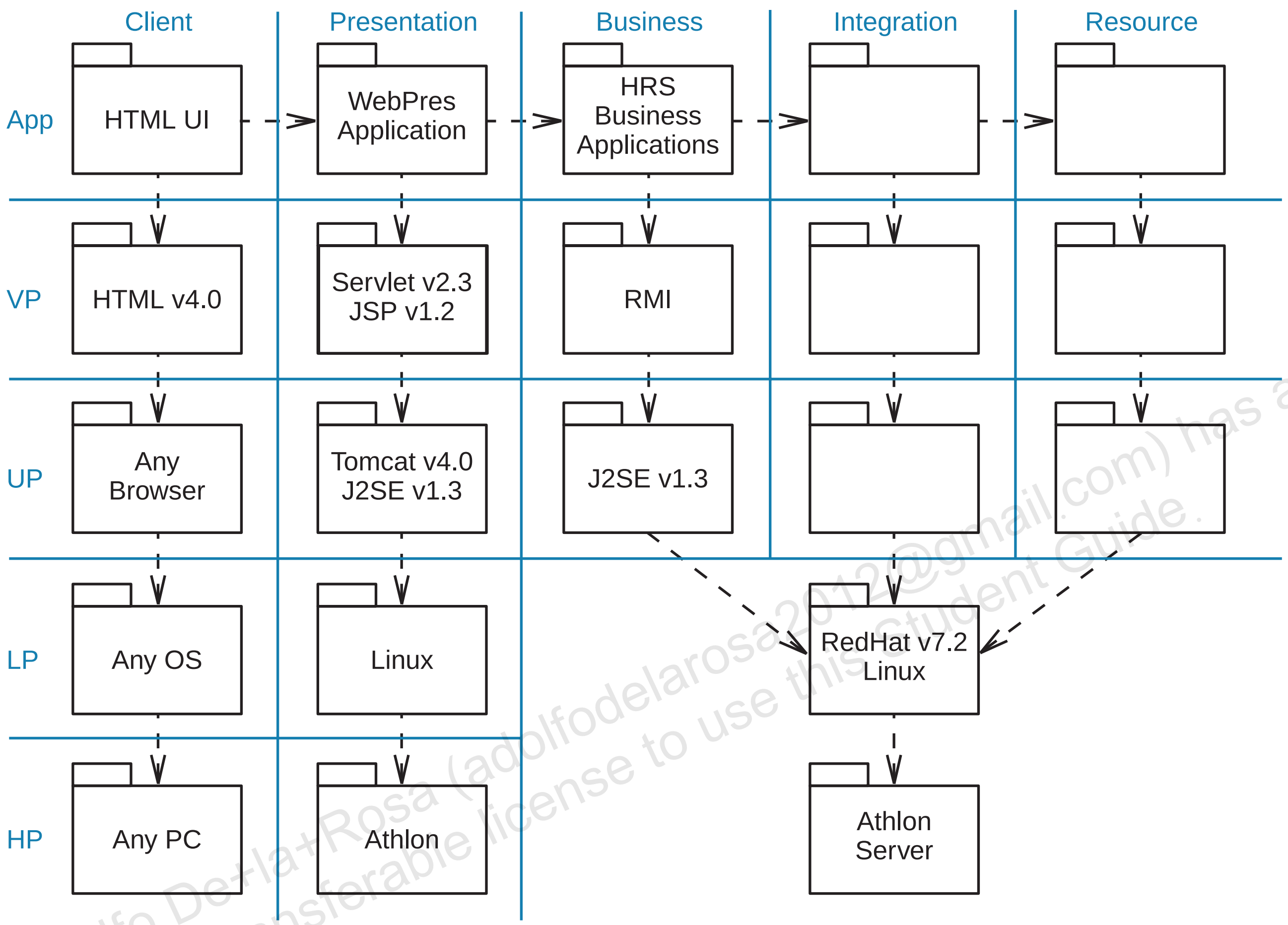


Figure 12-25 Tiers and Layers Diagram for the WebPresenceApp

Introducing the Resource and Integration Tiers

Exploring the Resource Tier

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

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

The Resource tier includes:

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

Exploring Object Persistence

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

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

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

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

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

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

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

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

Persistence Issues

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

- Type of data storage

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

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

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

- Data schema that maps to the Domain model

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

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

- Integration components

  There are roughly two types of integration components.

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

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

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

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

Creating a Database Schema for the Domain Model

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

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

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

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

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

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

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

Simplified HRS Domain Model

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

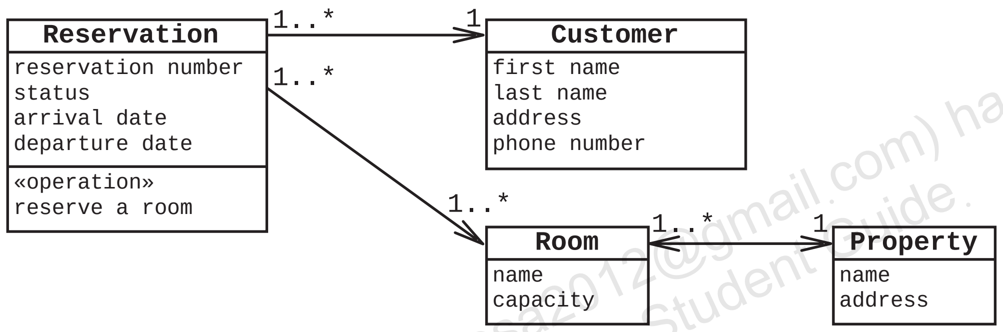


Figure 12-26 A Simplified Domain Model of the Hotel Reservation System

Step 1 – Map OO Entities to DB Tables

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

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

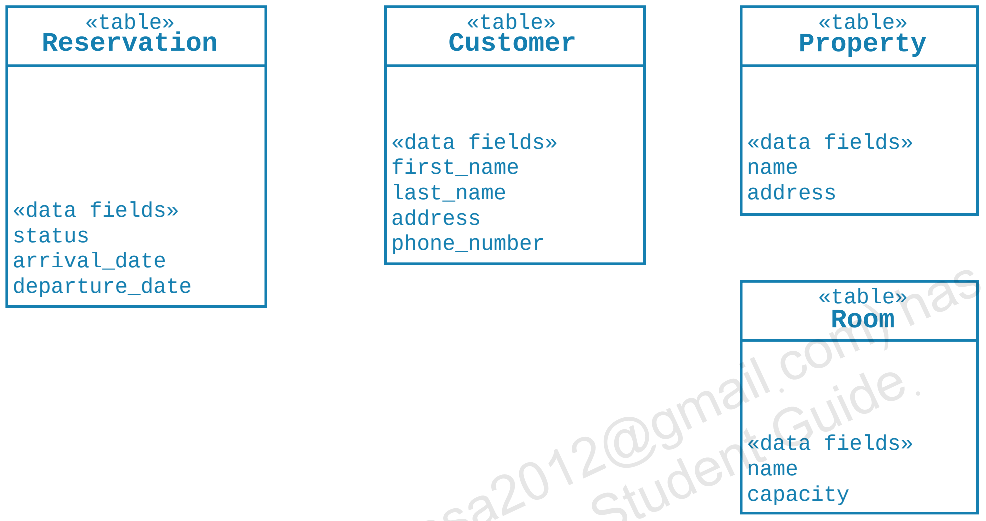


Figure 12-27 Step 1 – Creating Entity Tables

Step 2 – Specify the Primary Keys for Each Table

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

- Independent

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

- Dependent

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

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

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

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

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

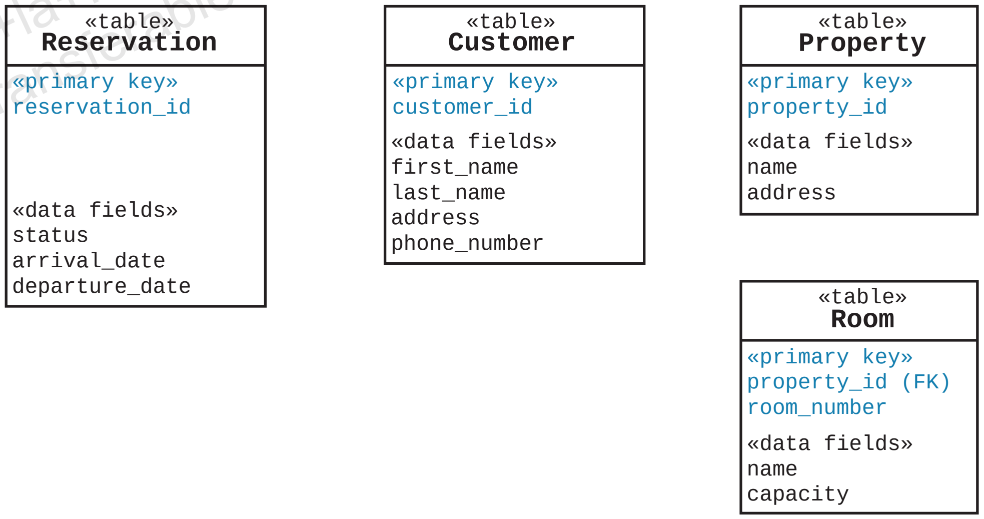


Figure 12-28 Step 2 – Specifying Primary Keys

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

Each association in the Domain model is represented by a foreign key relationship in the ER diagram. A foreign key is a field[3] within a table that refers to the primary key of the related table. For example, in the Domain model a reservation is related to a single customer. This relationship can be modeled in the database with a foreign key in the Reservation table to the Customer table (using the `customer_id` key field).

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

Figure 12-29 shows these relationships.

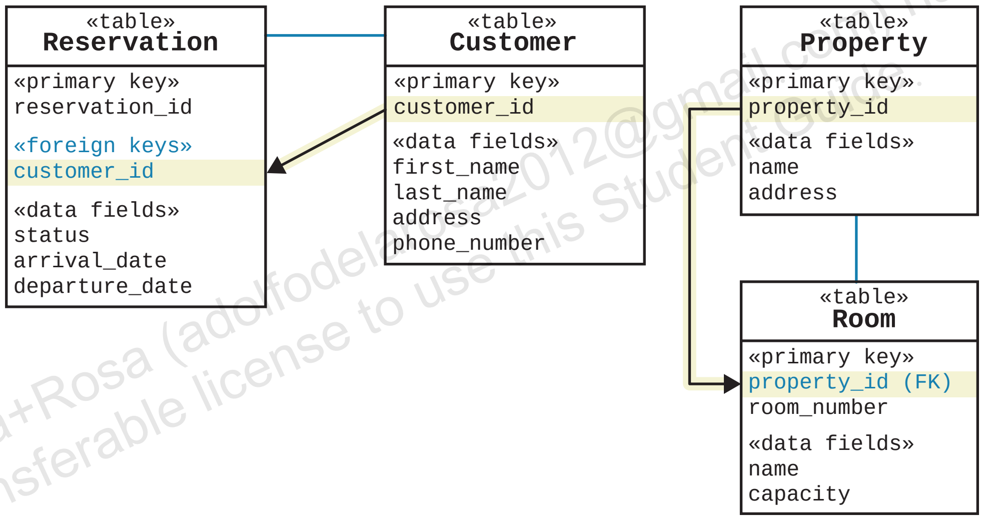


Figure 12-29 Step 3 – Creating One-to-Many Relationships

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

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

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

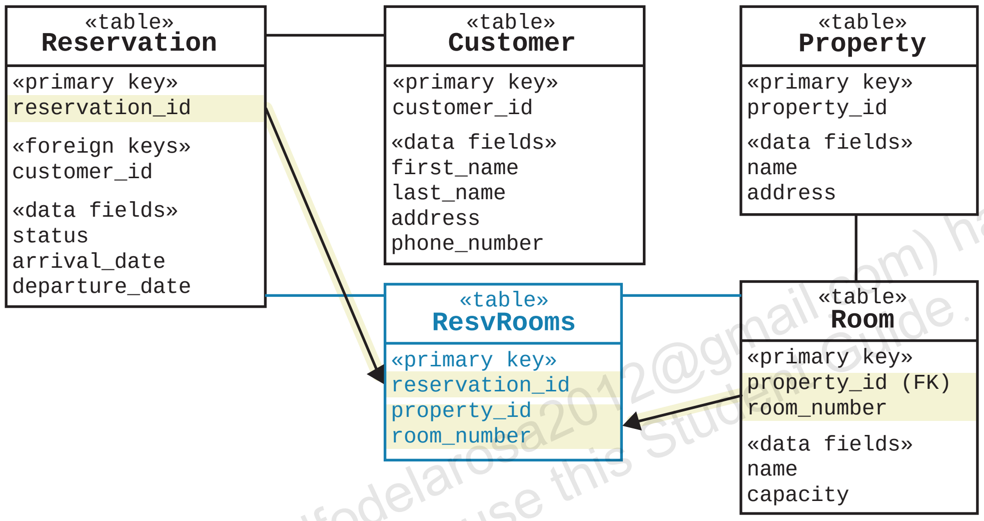

«table» Reservation
«primary key» reservation_id
«foreign keys» customer_id
«data fields» status arrival_date departure_date
«table» Customer
«primary key» customer_id
«data fields» first_name last_name address phone_number
«table» Property
«primary key» property_id
«data fields» name address
«table» ResvRooms
«primary key» reservation_id property_id room_number
«table» Room
«primary key» property_id (FK) room_number
«data fields» name capacity

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

The ResvRooms table usually[4] contains only the primary keys from each associated table. For example, `reservation_id` from the Reservation table and the compound key (`property_id`, `room_number`) from the Room table. This table will contain a row for each room in a given reservation, for all reservations.

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

Overview of the Detailed Deployment Diagram

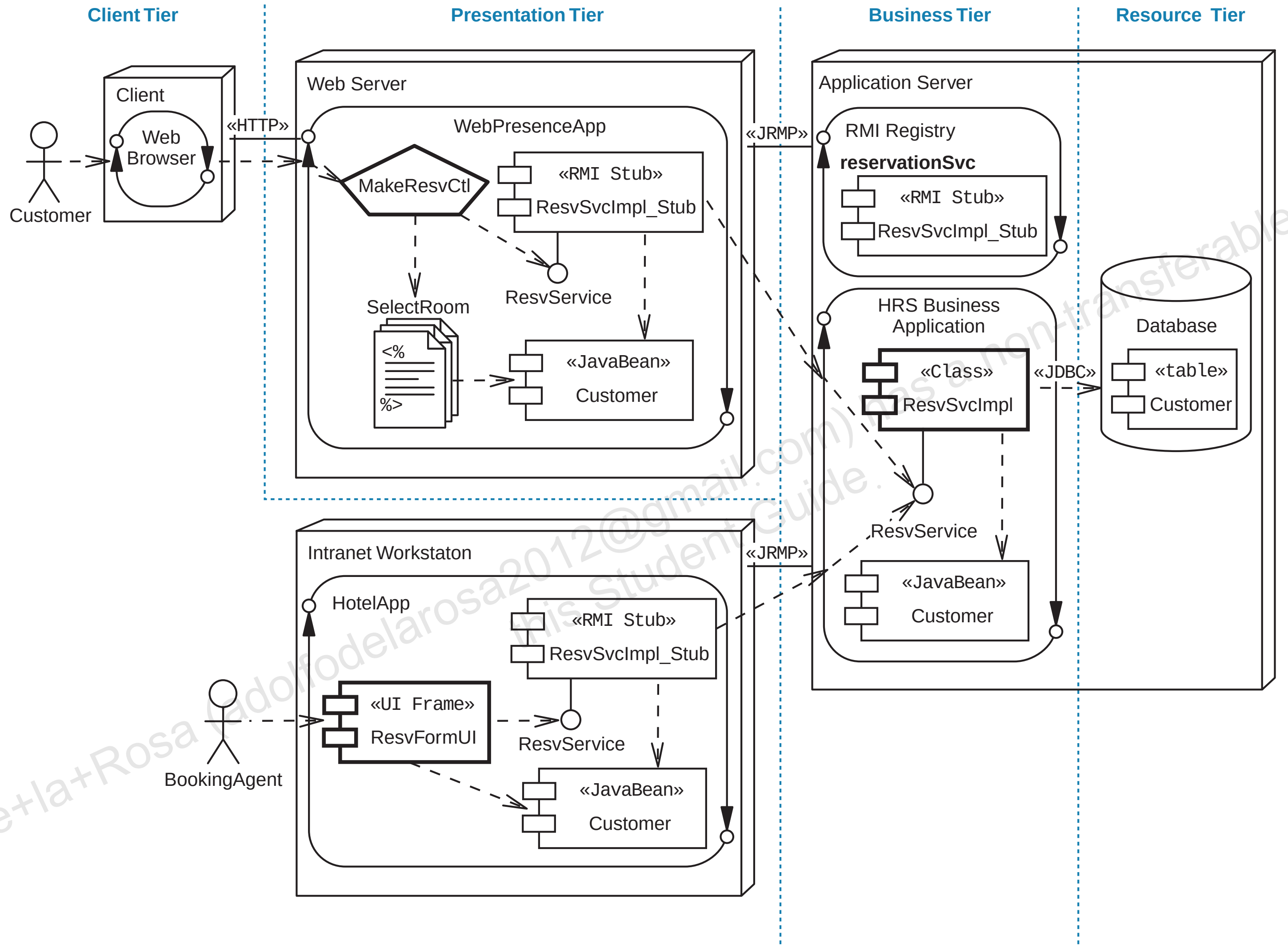


Figure 12-31 Example Detailed Deployment Diagram

Overview of the Tiers and Layers Package Diagram

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

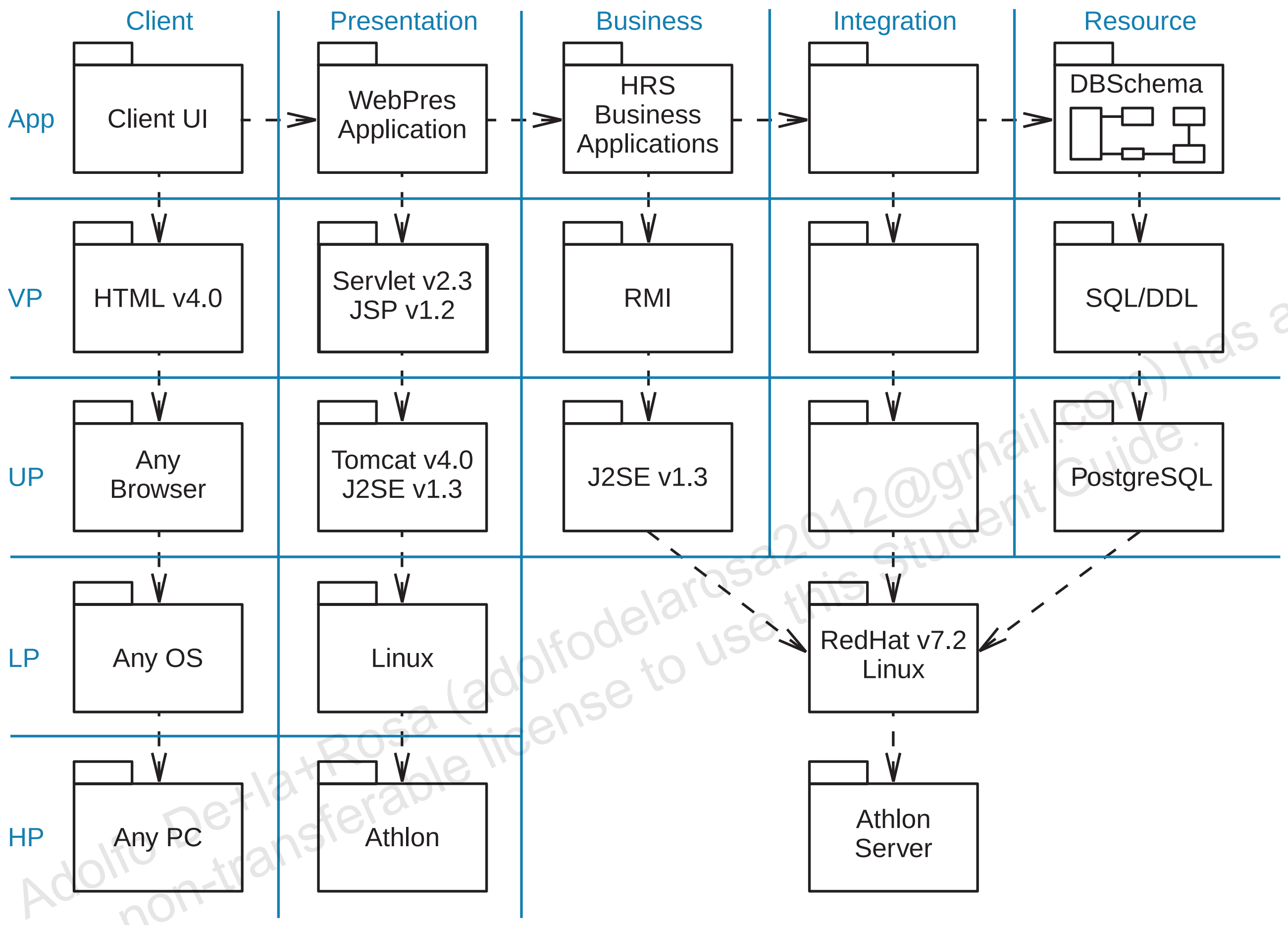


Figure 12-32 Partial HRS Tiers and Layers Package Diagram

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

Exploring Integration Tier Technologies

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

The Integration tier separates the entity components from the resources.

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

- Data sources

  A data source is a persistent object storage mechanism. As discussed in ''Exploring Object Persistence'' on page 12-33, there are many such mechanisms.

- Enterprise Information Systems (EIS)

  An EIS refers to any pre-existing software system that provides irreplaceable value to the entire company. These might be proprietary legacy systems or commercial products such as IBM's Customer Information Control System (CICS).

- Computational libraries

  A computation library is any software system that provides mathematical or simulation capabilities.

- Message services

  A message service is any tool that enables independent applications to communicate using asynchronous messages.

- Business to Business (B2B) services

  A B2B service is a service that integrates two or more applications across multiple companies.

This section only discusses integration with data sources.

DAO Pattern

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

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

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

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

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

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

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

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

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

Details of the DAO Pattern

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

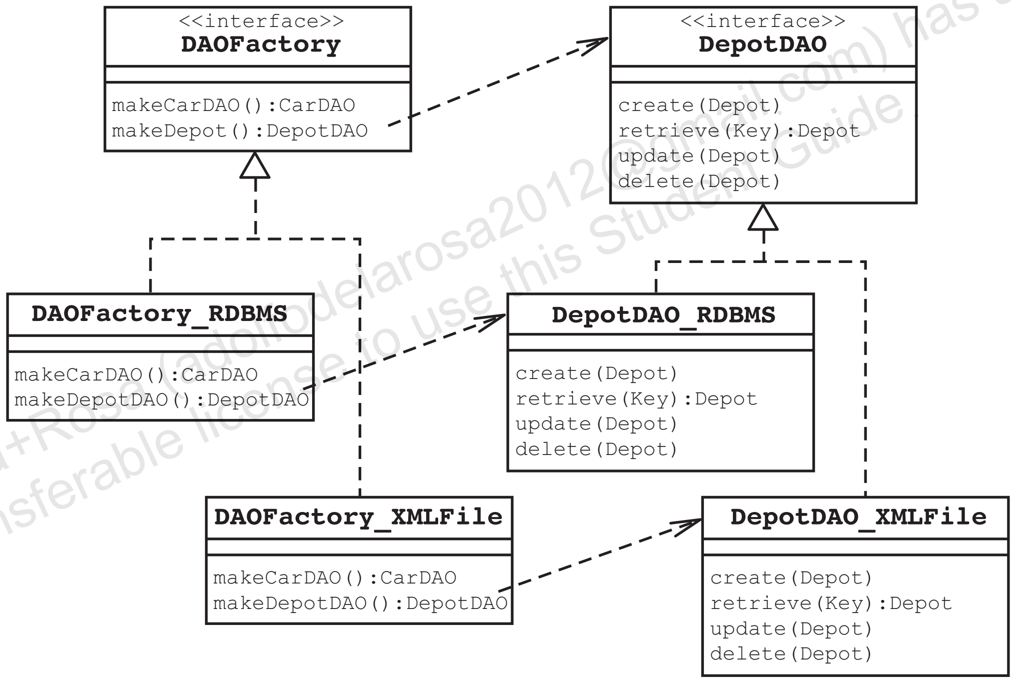


Figure 12-33 DAO Pattern Example 1

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

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

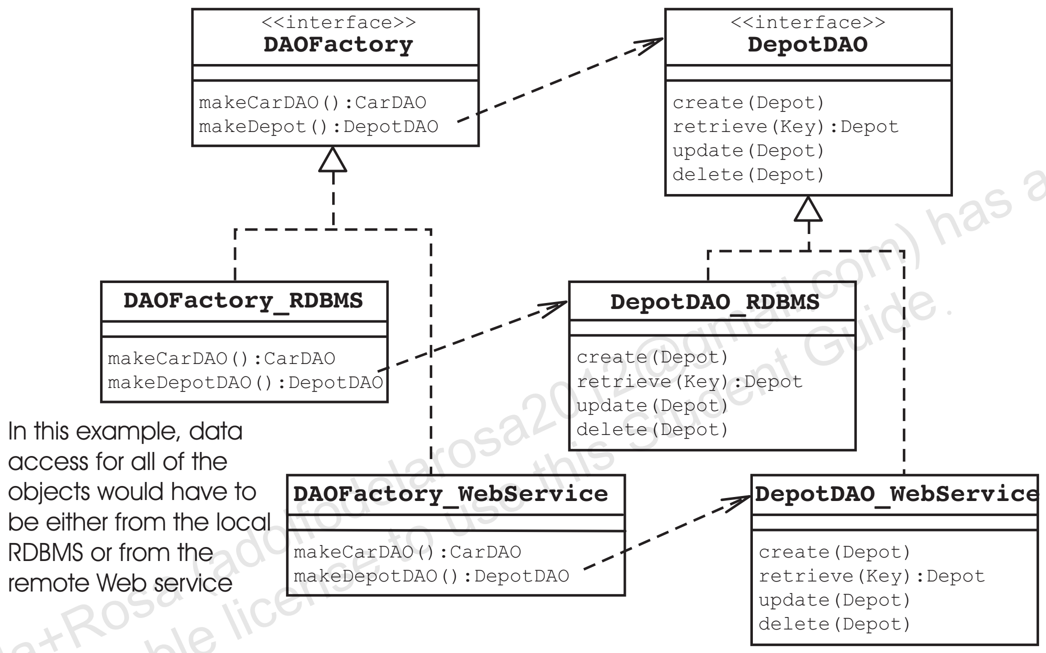


Figure 12-34 DAO Pattern Example 2

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

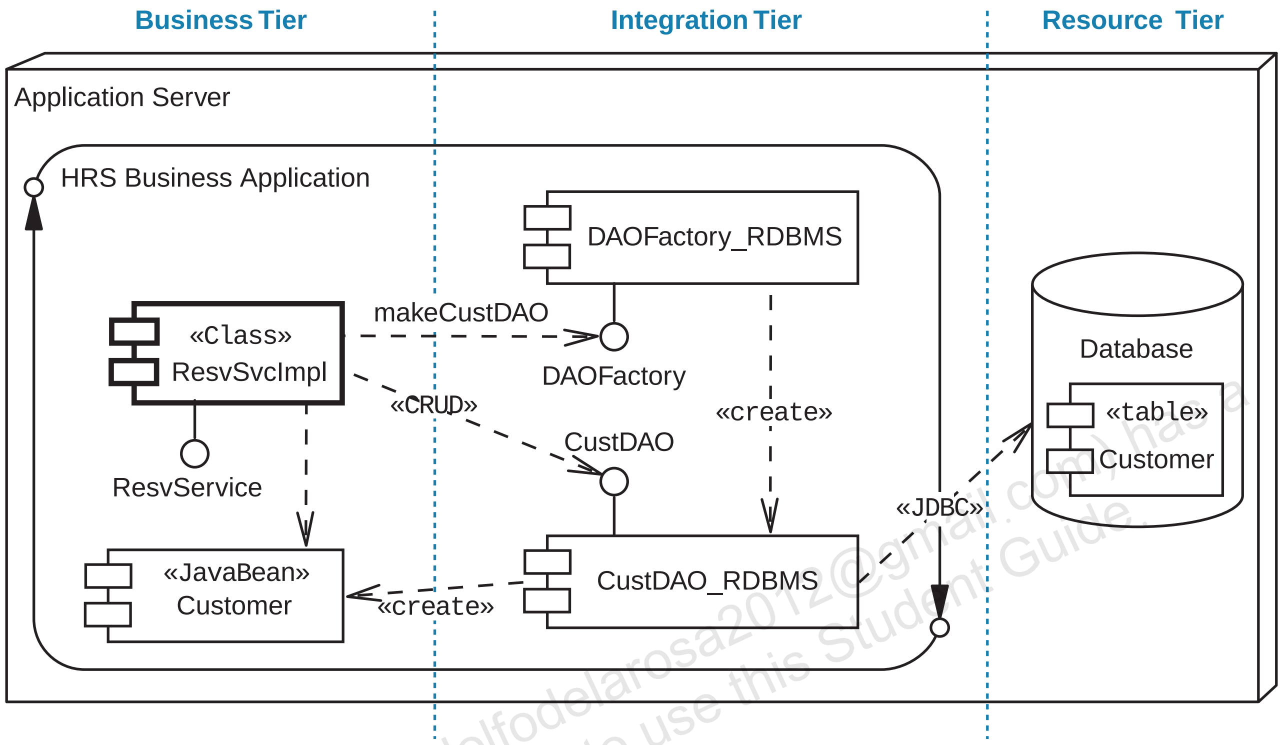


Figure 12-35 The DAO Pattern in a Detailed Deployment Diagram

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

Overview of the Detailed Deployment Diagram

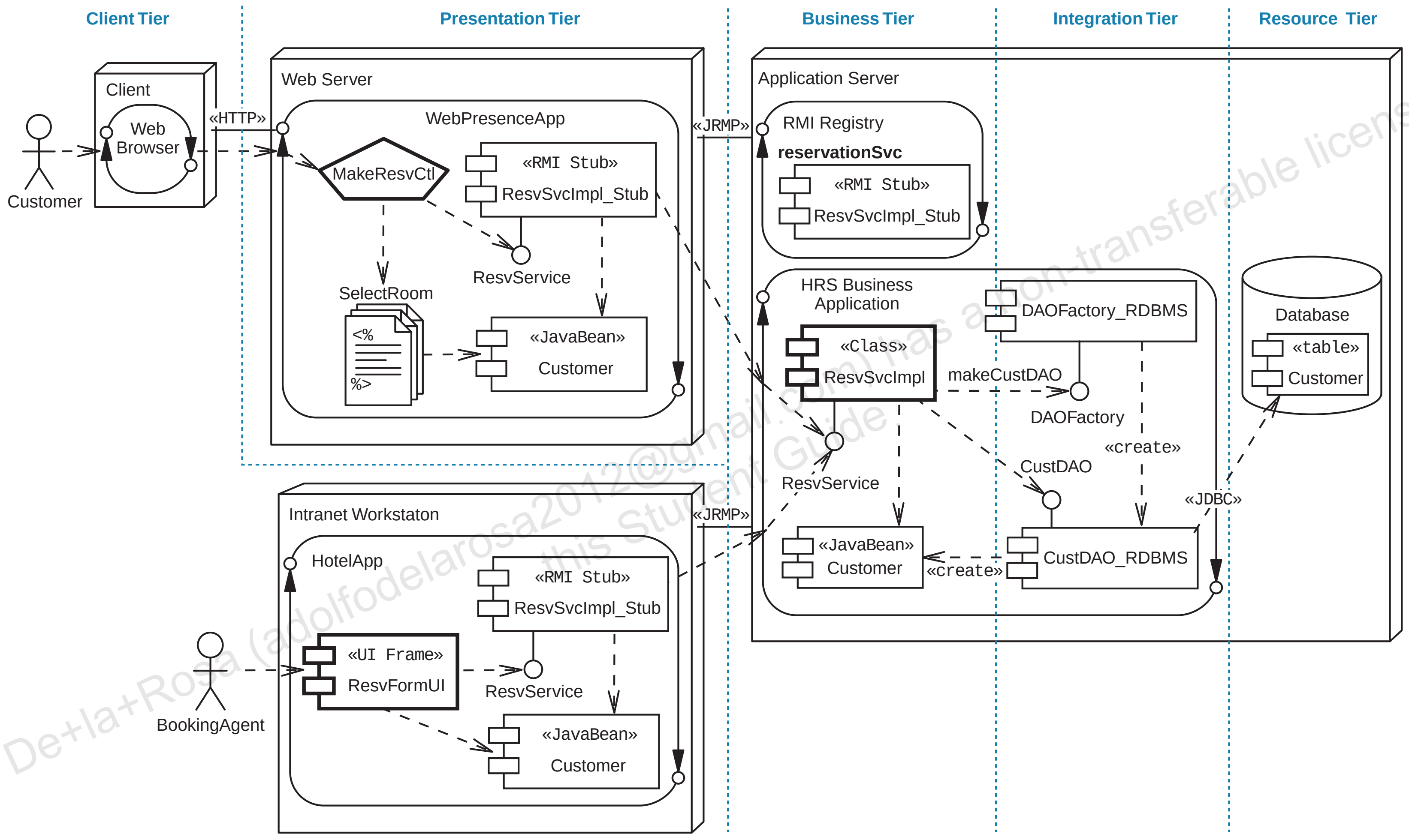


Figure 12-36 Example Detailed Deployment Diagram

Overview of the Tiers and Layers Package Diagram

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

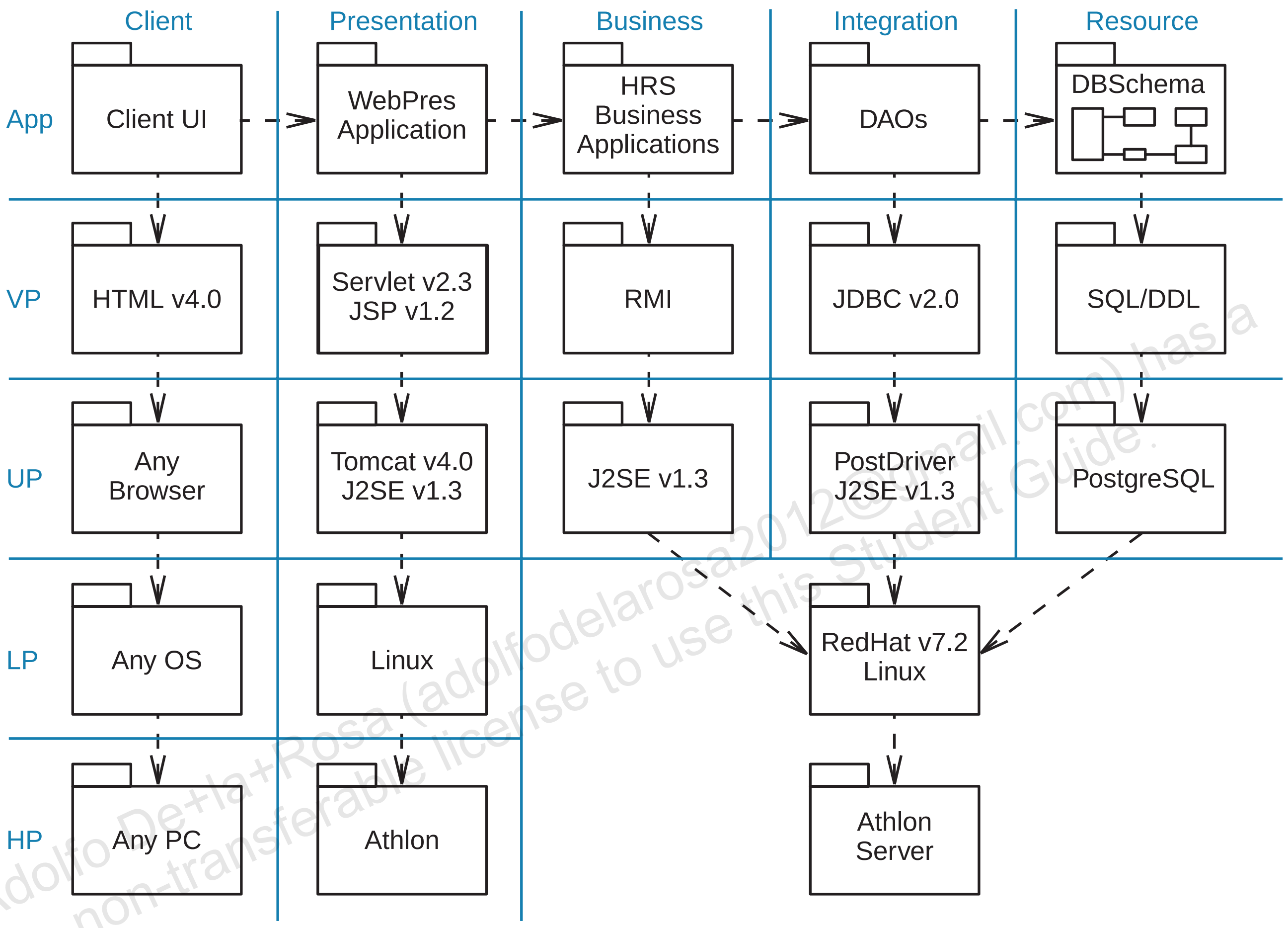


Figure 12-37 Complete HRS Tiers/Layers Package Diagram

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

Java™ Persistence API

Java Persistence API:

- Is an alternative to DAO

  Although the Java Persistence API is an alternative to using DAOs, it may be possible to use legacy DAOs to use the Java Persistence API. Therefore, you do not have to change your existing business logic components.
- Draws on the best ideas from alternative persistence technologies such as Hibernate, TopLink, and Java Data Objects (JDO)
- Uses a persistence provider such as Hibernate or TopLink
- Is a Plain Old Java Object (POJO) persistence API for Object/Relational mapping

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

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

- Uses an entity manager to manage objects

Java Persistence API Entity Manager

A Java Persistence API entity manager can be requested to:

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

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

Introducing the Solution Model

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

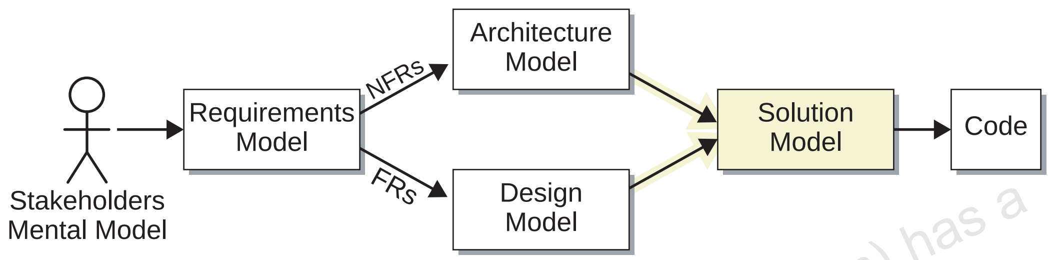


Figure 12-38 Introducing the Solution Model

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

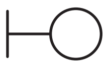
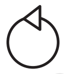
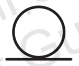
A Solution model can be represented by actual code; for example, if the architecture team built an Architecture baseline. The structure of this code set can act as a Solution model for the development team.

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

Overview a Solution Model for GUI Applications

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

Table 12-1 Standard Analysis Component Icons

Component Type	Icon
Boundary	
Service	
Entity	

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

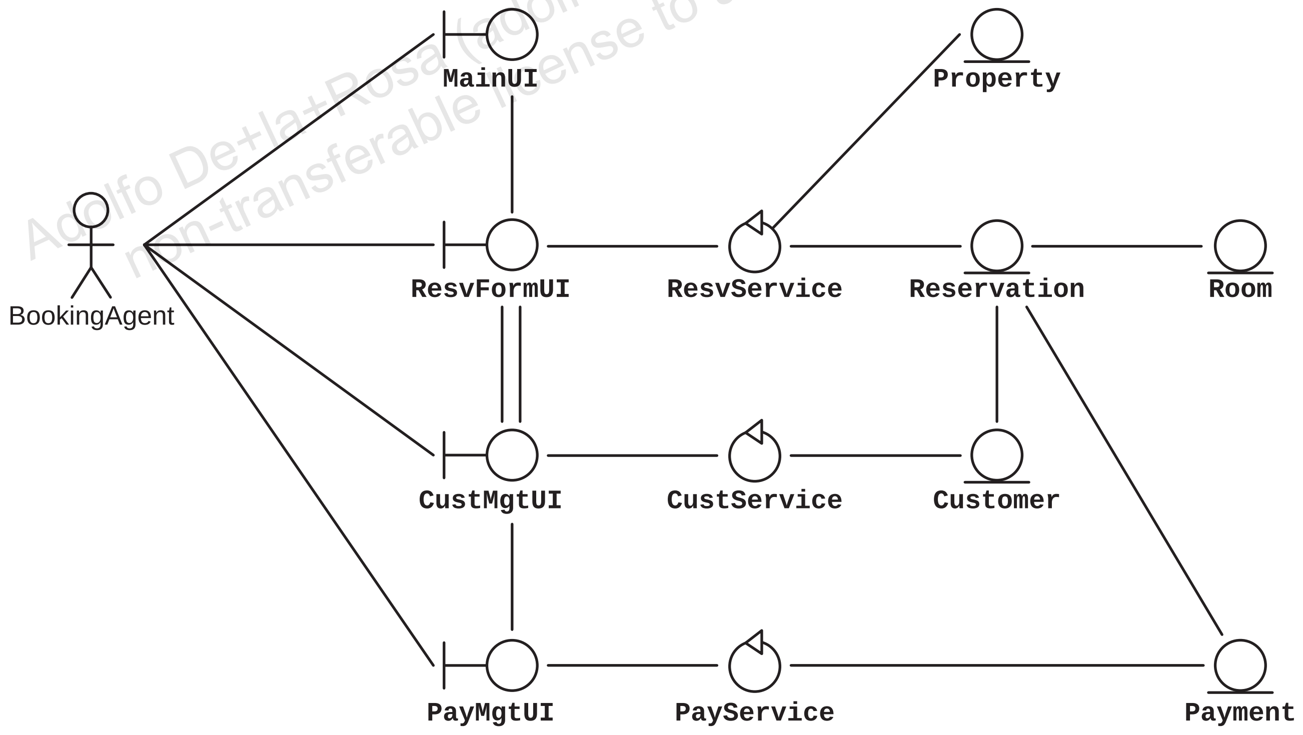


Figure 12-39 A Complete Analysis Model for the Create a Reservation Use Case

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

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

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

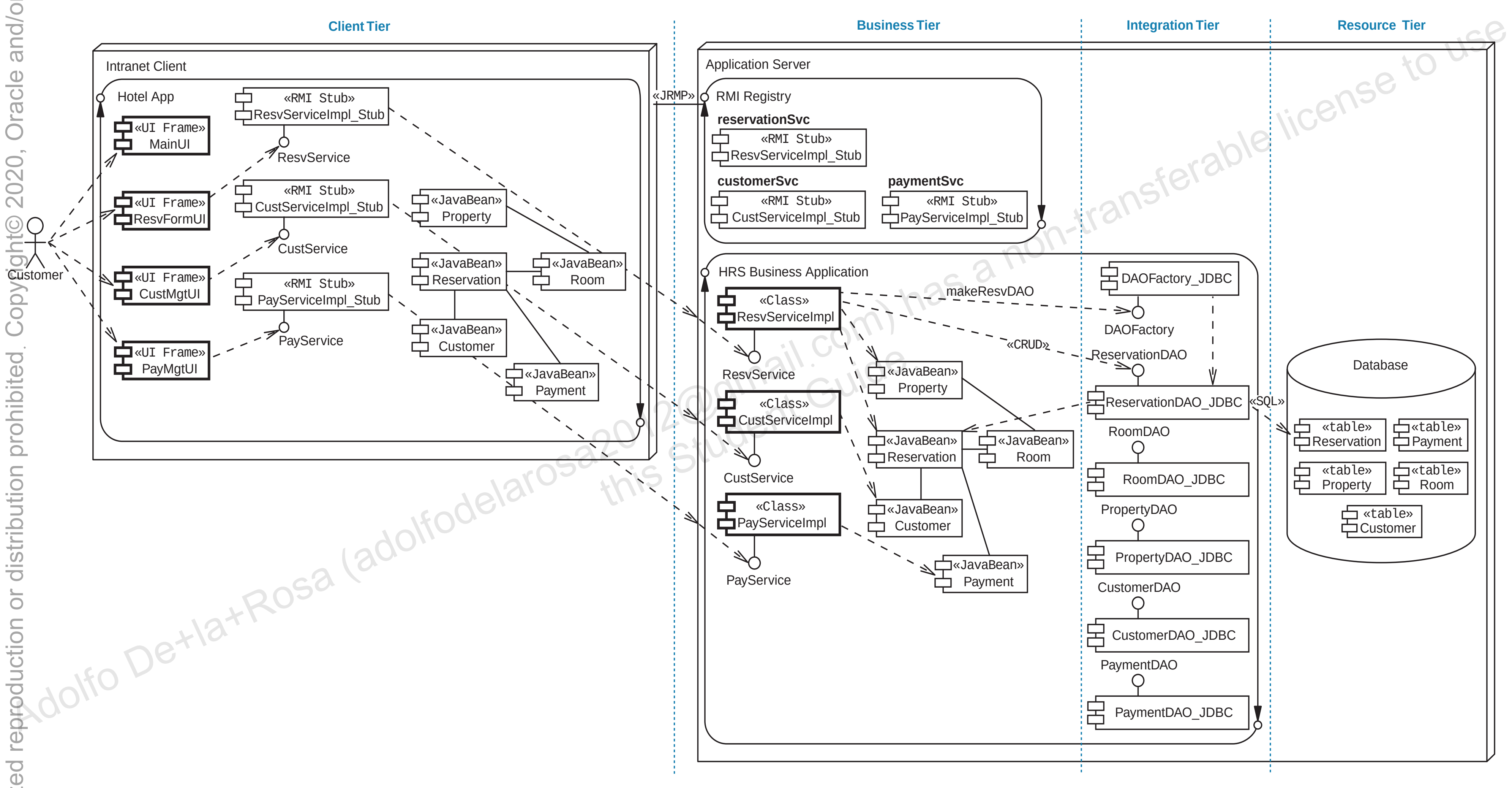


Figure 12-40 A Complete Solution Model for the Create a Reservation Use Case

Overview a Solution Model for WebUI Applications

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

Table 12-2 WebUI Analysis Component Icons

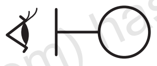
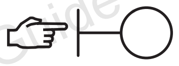
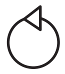
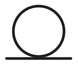
Component Type	**Icon**
View	
Controller	
Service	
Entity	

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

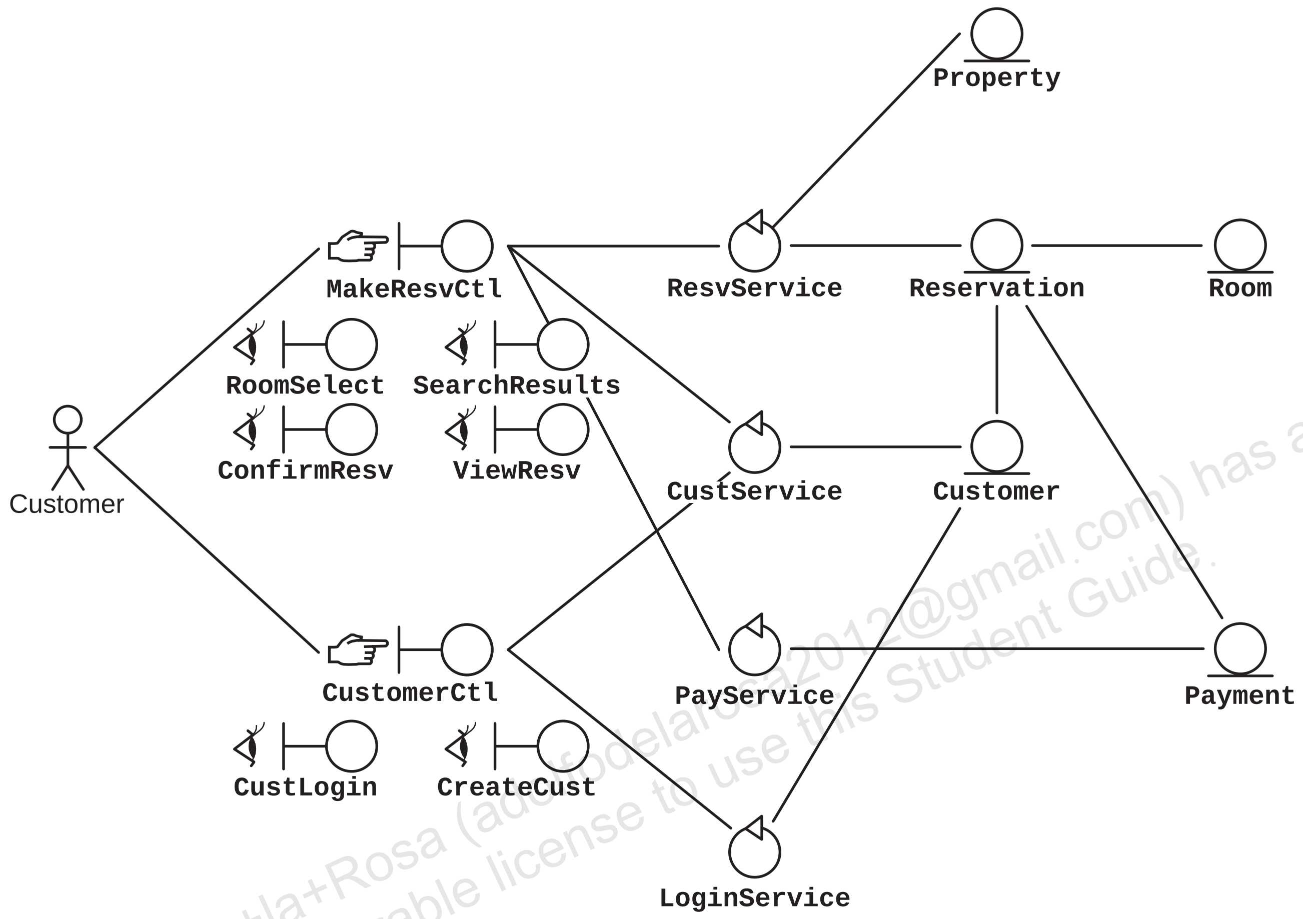


Figure 12-41 A Complete Analysis Model for the Create a Reservation Online Use Case

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

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

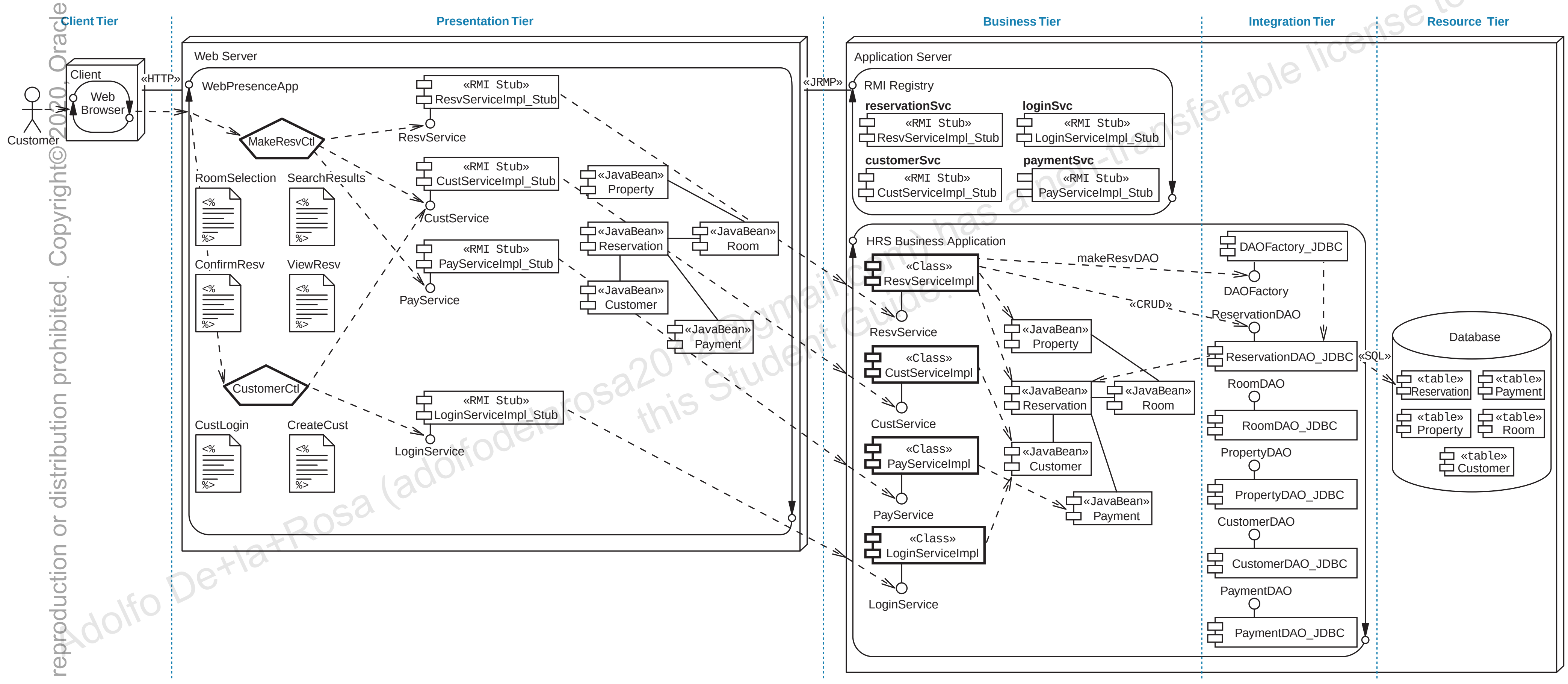


Figure 12-42 A Complete Solution Model for the Create a Reservation Online Use Case

Summary

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

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

Module 13

Refining the Class Design Model

Objectives

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

- Refine the attributes of the Domain model
- Refine the relationships of the Domain model
- Refine the methods of the Domain model
- Declare the constructors of the Domain model
- Annotate method behavior
- Create components with interfaces

Additional Resources

Additional resources – The following reference provides additional information on the topics described in this module:

- The Object Management Group. “Unified Modeling Language (UML), Version 2.2” [`http://www.omg.org/technology/documents/formal/uml.htm`].
- 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.

Process Map

This module covers the next step in the Design workflow: refining the Design model. Figure 13-1 shows the activity and artifacts discussed in this module.

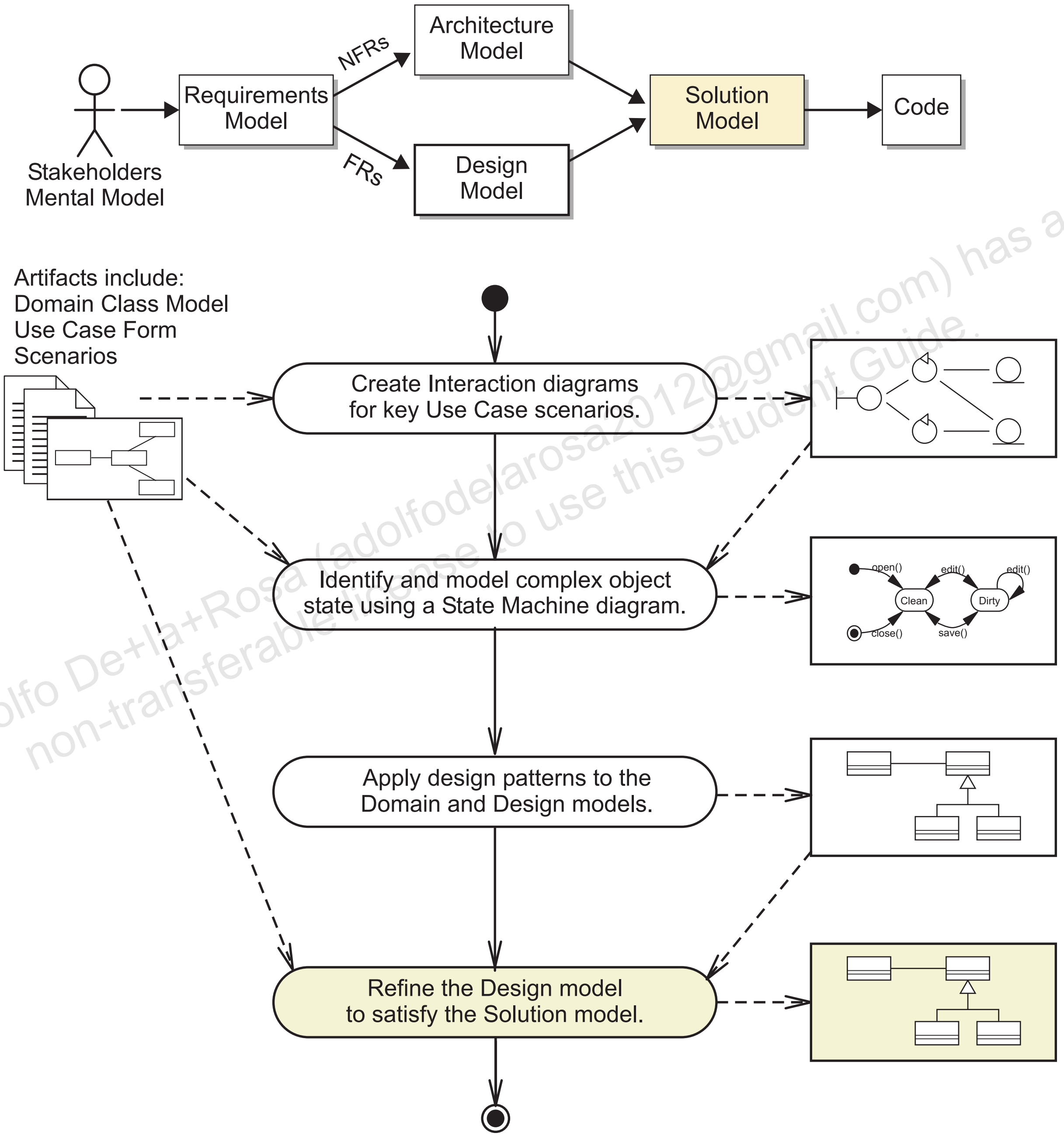

Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code
Artifacts include:
Domain Class Model
Use Case Form
Scenarios
Create Interaction diagrams for key Use Case scenarios.
Identify and model complex object state using a State Machine diagram.
open()
edit()
edit()
Clean
Dirty
close()
save()
Apply design patterns to the Domain and Design models.
Refine the Design model to satisfy the Solution model.

Figure 13-1 Class Design Workflow Process Map

Refining Attributes of the Domain Model

This section discusses several issues around refining the attributes of the domain entities from the Analysis workflow.

Refining the Attribute Metadata

An attribute declaration in UML Class diagrams includes the following:

- Name

  This is the only mandatory feature of an attribute element. This declares the name of the attribute. During Analysis, you can specify attribute names with spaces in them; for example, “first name.” However, during Design you should convert the attribute names to a style that is appropriate to the implementation language; for example, “firstName” would be appropriate using the Java programming language.
- Visibility

  This optional feature defines how visible (or accessible) an attribute is outside of the class. Table 13-1 shows there are four predefined visibility values. The default value is public.

Table 13-1 UML Visibility Indicators

Visibility	**Indicator**
Public	+
Protected	#
Package private	~
Private	-

- Type

  This is the data type of the attribute. This is often language-specific, but most language support primitive values, such as integers, floating point numbers, and Boolean values, as well as object references.
- Multiplicity

  This specifies how many values might exist for the attribute. By default, the multiplicity is one.

- Initial value

  This specifies the initial value of this attribute when an object of this class is constructed. This is the default value at initialization, but it can be overridden by a constructor.

- Constraint (one of `changeable`, `addOnly`, or `frozen`)

  This optional feature specifies the characteristic of *changeability* for the attribute at runtime. The value `changeable` means that the attribute can be changed at any time. The value `frozen` means that the attribute must be set during object instantiation and cannot be reset. The value `addOnly` applies to attributes that have a multiplicity greater than one; it means that values can be added to this attribute but no values can be removed.

  These constraints are defined by the UML specification. However, you can use stereotypes to apply additional semantics to the attribute.

The UML syntax for an attribute declaration is as follows:

```
[visibility] name [multiplicity] [: type] [= init-
value] [{constraint}]
```

Figure 13-2 illustrates a refinement of the `Customer` class in which the attributes are fully specified. In this design, the first and last names of a customer cannot be changed after the object is created. Also the `phone` attribute supports one or two `PhoneNumber` objects: one for the customer's home and one for the work phone number.

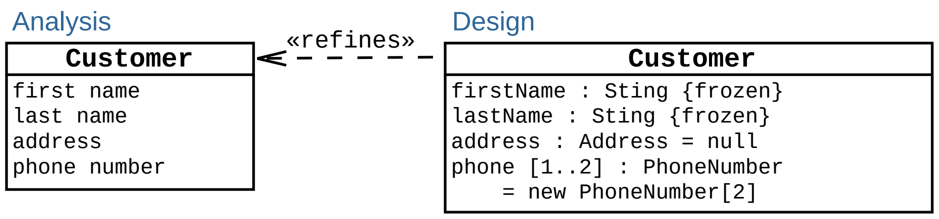


Figure 13-2 Example Refinement of Attributes

Choosing an Appropriate Data Type

Choosing a data type is a trade-off of:

- Representational transparency

  How well does the data type correspond to the programmer's mental model of the attribute?
- Computational time

  How much time is required to convert the data into the representations needed for the user interface or data storage?
- Computational space

  How much space (in bytes) does the data require in memory? This issue is very important for small devices like cellphones and PDAs.

Table 13-2 illustrates a few of the issues in selecting a data type to represent a phone number attribute.

Table 13-2 Choosing a Data Type for a Phone Number

Data Type	Discussion
`String`	This data type might require mapping between the UI representation and the storage (DB) representation.
`long`	This data type conserves space, but might not be sufficient to represent large phone number (such as international numbers).
`PhoneNumber`	A value object is a class that represent the phone data. This data type is representationally transparent, but requires additional coding.
`char` array	This data type is similar to a `String` and adds no value.
`int` array	This data type conserves space, but is not represenationally transparent.

Creating Derived Attributes

In Analysis, you might have an attribute that you know can be derived from another (more stable) source. The canonical example is the calculation of a person's age from their date of birth. A derived attribute is represented with a "/" character in front of the attribute name. Figure 13-3 illustrates this.

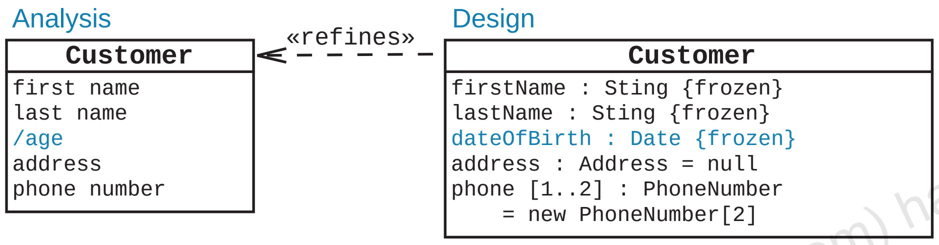


Figure 13-3 Example Derived Attribute

Applying Encapsulation

To use encapsulation, follow these steps:

1. Make all attributes private (visibility).
2. Add public accessor methods for all readable attributes.
3. Add public mutator methods for all writable (non-frozen) attributes.

Figure 13-4 shows an encapsulated class.

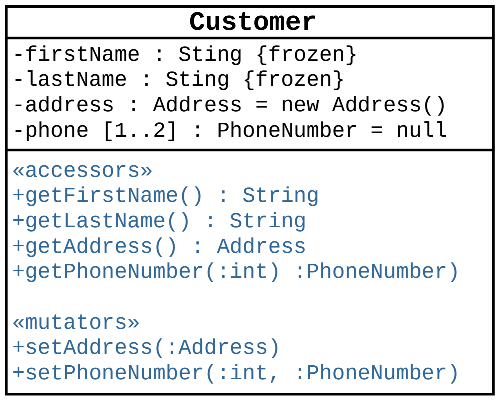

Customer
-firstName : Sting {frozen}
-lastName : Sting {frozen}
-address : Address = new Address()
-phone [1..2] : PhoneNumber = null
«accessors»
+getFirstName() : String
+getLastName() : String
+getAddress() : Address
+getPhoneNumber(:int) :PhoneNumber)
«mutators»
+setAddress(:Address)
+setPhoneNumber(:int, :PhoneNumber)

Figure 13-4 Example Use of Encapsulation

Refining Class Relationships

There is no clear distinction between Analysis and Design especially in regards to modeling class relationships.

Design usually addresses these details:

- Type: association, aggregation, and composition
- Direction of traversal (also called navigation)
- Qualified associations
- Declaring association management methods
- Resolving many-to-many associations
- Resolving association classes

Relationship Types

There are three types of relationships:

- Association
- Aggregation
- Composition

These relationships imply that the related object is somehow tied to the original object (usually as an instance attribute).

Note – There is another type of relationship, Dependency, which states that one object uses another object to do some work, but that there is no instance attribute holding that object. For example, if `Class1` has a method that has a parameter of type `Class2`, then you can say that `Class1` depends on `Class2`.

Association

"The semantic relationship between two or more classifiers that specifies connections among their instances." (OMG page 537)

An association is a relationship in which one object holds a reference to another object as an instance variable. Figure 13-5 illustrates an example association.

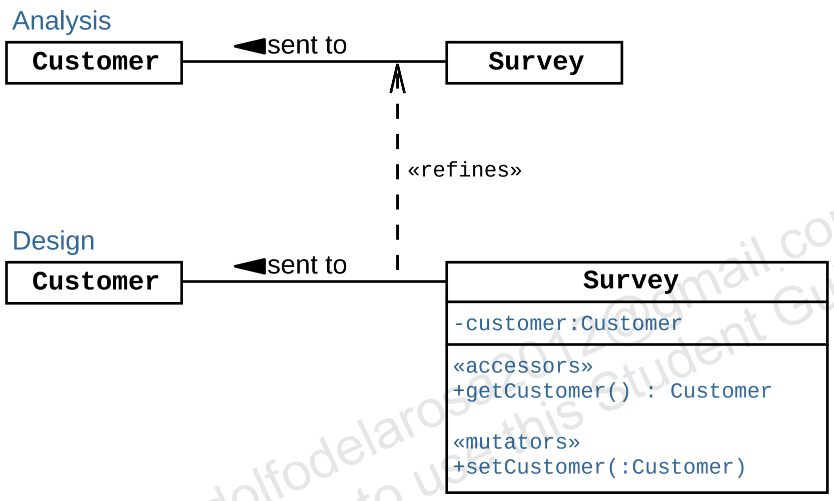


Figure 13-5 An Association Example

Aggregation

> "A special form of association that specifies a whole-part relation between the aggregate (whole) and a component part." (OMG page 537)

The distinction between an association and an aggregation is largely semantic; there is no functional difference between the implementation of these two relationship types. Figure 13-6 illustrates an example aggregation.

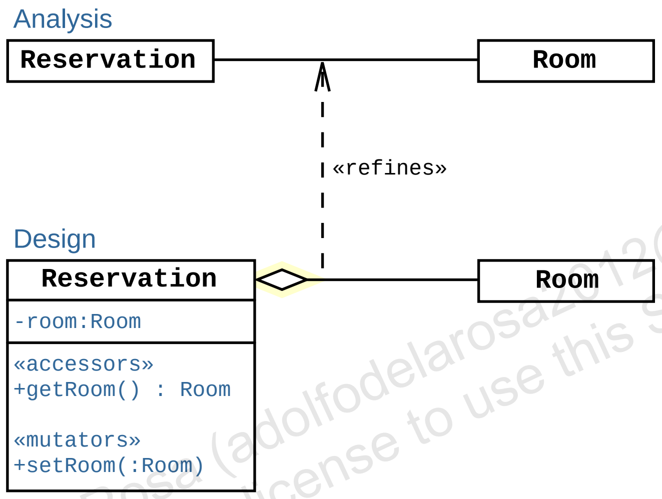


Figure 13-6 An Aggregation Example

Composition

"A form of aggregation that requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts." (OMG page 540)

The distinction between an aggregation and a composition is semantically similar, but functionally different. In general, the parts of a composition are not accessible outside of the whole object. For example, it might be important to model the payment for a reservation as a composition in which the Reservation object is in complete control over the creation (and destruction) of the Payment object. To modify the information about a payment, the Reservation class must have methods to modify the data within the Payment object. Figure 13-7 illustrates this example.

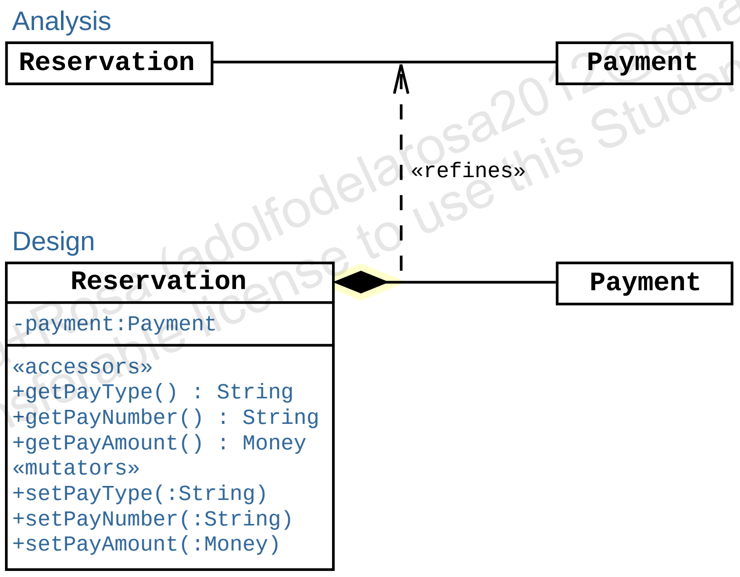


Figure 13-7 A Composition Example

This rule of not exposing the part object outside of the whole object might be impractical in some situations. For example, if the part object has many attributes which need to be modified by a client object outside of the whole object, then it would be impractical to create an accessor and mutator method for every part attribute. Also, if the composition is of many parts, then the accessor and mutator methods would have to use a qualifier to distinguish which part is to be accessed. Use of a qualifier can be difficult. Alternatively, you could expose the part objects to the client object, but that risks having the part object used inappropriately by the rest of the system. This decision is a design trade-off.

Navigation

A navigation arrow shows the direction of object traversal at runtime. Figure 13-8 illustrates there are three fundamental forms of navigation.

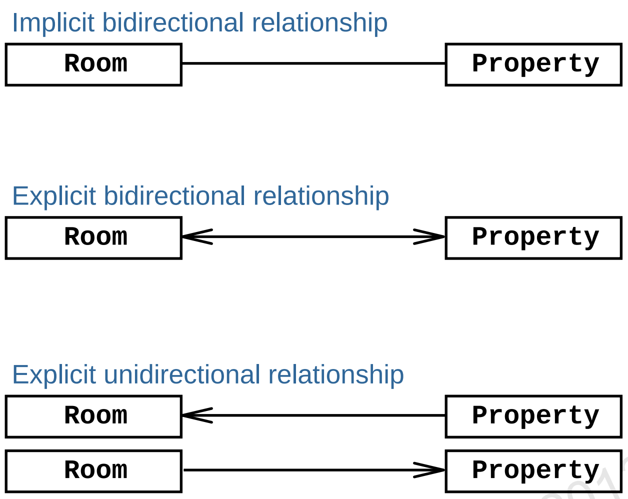

Implicit bidirectional relationship
Room
Property
Explicit bidirectional relationship
Room
Property
Explicit unidirectional relationship
Room
Property
Room
Property

Figure 13-8 The Three Forms of Navigation Indicators

Navigation implies having accessor methods. In a bidirectional relationship both classes will have accessor methods. For example, in the first two cases in Figure 13-8 the `Room` class would have a `getProperty` method and the `Property` class would have a `getRoom` method. In a unidirectional relationship only the class at the end of the relationship would have an accessor method. For example, the third case in Figure 13-8, the `Property` class would have a `getRoom` method, but the `Room` class would *not* have a `getProperty` method.

Sometimes during analysis, you do not know what direction the software will need to navigate the association. This problem should be resolved during design. For example, the business analyst might not have determined if a `Payment` object could be navigated back to its corresponding `Reservation` object. Therefore, the analyst left the relationship as implicitly bidirectional. However, through Robustness analysis the design team decided that it is only necessary to navigate from the `Reservation` object to the `Payment` object. Figure 13-9 illustrates the appropriate navigation they assign.

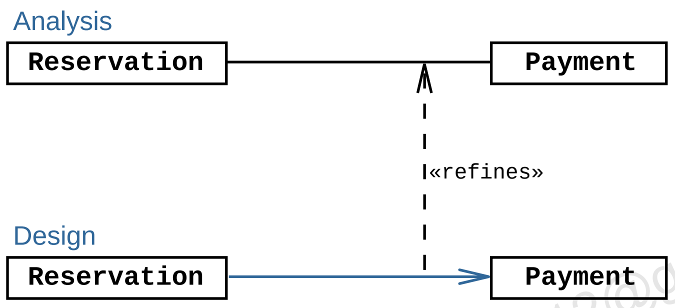


Figure 13-9 An Example Navigation Refinement

Qualified Associations

"An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association." (OMG page B-15)

In one-to-many or many-to-many associations, it is often useful to model how the system will access a single element in the association. To do this you must define a *qualifier* that provides a datum that uniquely identifies the particular object. For example, a hotel property consists of multiple rooms, and each room has a unique name. Figure 13-10 illustrates this name can be used to qualify the association.

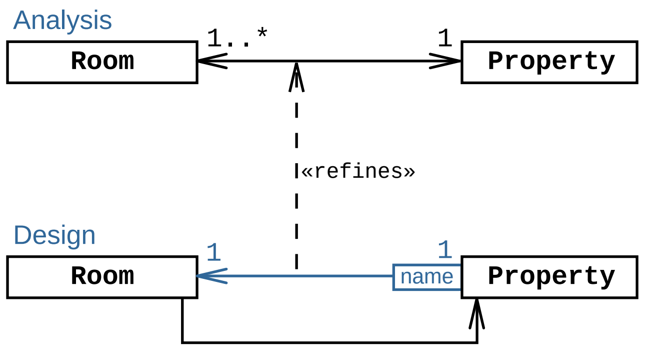


Figure 13-10 An Example Qualified Association

Relationship Methods

Association methods enable the client to access and change associated objects. There are three cases: one-to-one, one-to-many, and many-to-many.

One-to-one relationships use a single instance variable reference. For example, the hotel management would like to send out surveys to customers. So the `Survey` class keeps an association to the `Customer` class using an instance variable and accessor and mutator methods. Figure 13-11 illustrates this example.

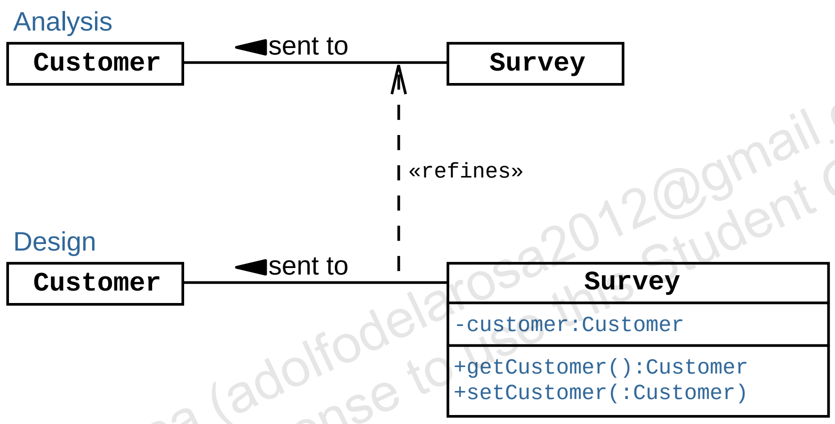


Figure 13-11 Association Methods for a One-to-One Relationship

One-to-many relationships require the use of collections. For example, a hotel property contains multiple rooms. The `Property` class would store the `Room` objects in a `Collection` object. The accessor method `getRooms` might return an `Iterator` object which would enable the client object to iterate over each element in that collection. The `addRoom` and `removeRoom` mutator methods modify the collection of rooms. Figure 13-12 illustrates this example.

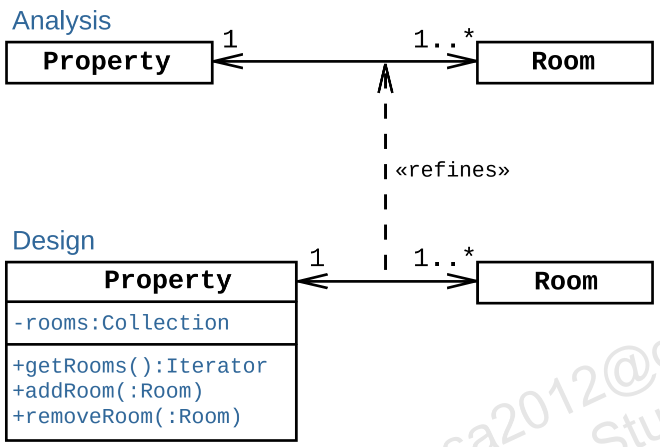


Figure 13-12 Association Methods for a One-to-Many Relationship

---

Note – The `Collection` and `Iterator` interfaces are built into the J2SE platform. Other languages have similar mechanisms for working with collections (also called containers).

Resolving Many-to-Many Relationships

Managing many-to-many relationships is challenging. This section describes two techniques for simplifying this type of relation.

First, consider dropping the many-to-many requirement during design. For example, an automotive software system might need to model people and the cars they own. Clearly, in analysis you might decide that any person can own multiple cars and any car might be own by multiple people (at different times). However, if the software you are creating is for a car dealership, then it is not important to record the history of owners for a given car; the system just needs to record who bought the car. In this situation there is no requirement for a many-to-many relationship, so during design the team decides to drop this requirement. Figure 13-13 illustrates this situation.

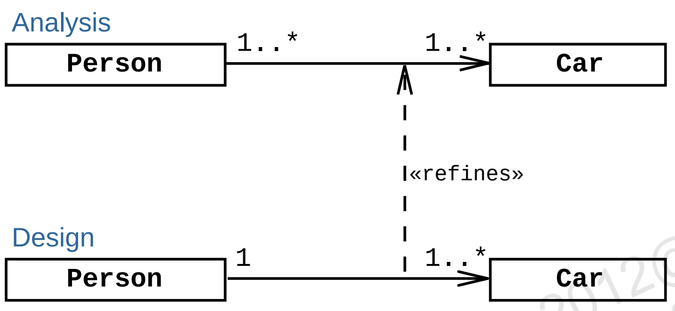


Figure 13-13 Drop the Many-to-Many Relationship

If the many-to-many association must be preserved, you can sometimes add a class in between the two classes that reduces the single many-to-many association to two one-to-many associations. In our person-to-car example, if the software needs to record the complete ownership history for a car, then an additional class, `Ownership`, could be introduced to resolve the many-to-many relationship. Figure 13-14 illustrates this situation.

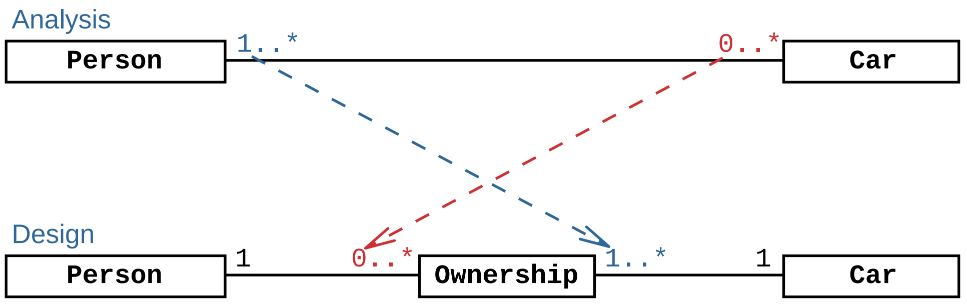


Figure 13-14 Introduce an Intermediate Relationship

Resolving Association Classes

An association class can only exist in the Analysis version of the Domain model. An association class is purely conceptual, there is no programming technique to directly implement this type of concept. Therefore, association classes should be resolved by some construct that can be implemented during Design.

Imagine a soccer application in which the system must model a game played by two teams. Relative to a game, each team would have a score and a flag indicating whether or not that team had to forfeit (for example, not enough players showed up to play the game). During analysis, this situation is nicely modeled with an association class called `TeamScore`. However, this association class must be resolved during design as most OO languages do not support association classes. The developers may decide to move the data for the results into the `Game` class. This solution is feasible because there is a definite number of teams for the game: two. Figure 13-15 illustrates this example.

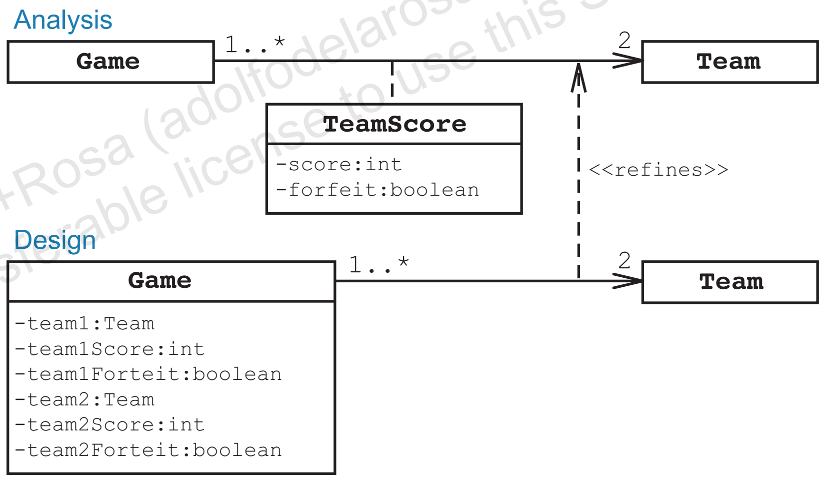


Figure 13-15 Resolving Association Classes: Case One

An alternative strategy is to place the association class in between the two primary classes. Figure 13-16 illustrates this.

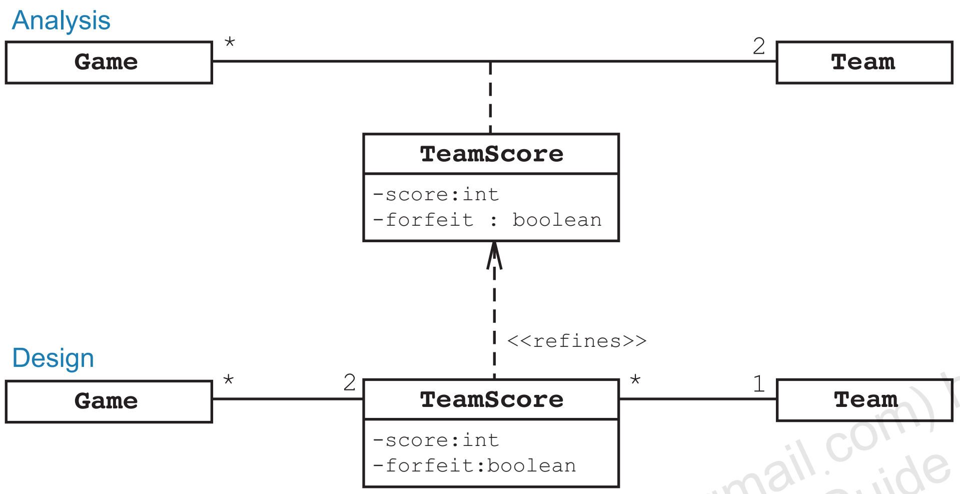


Figure 13-16 Resolving Association Classes: Case Two

Refining Methods

Methods are identified during the following activities:

- CRC analysis, which determines responsibilities
- Robustness analysis, which identifies methods in Service classes
- Design, which identifies accessor and mutator methods for attributes and associations

Other types of methods:

- Object management

  This type of method performs tasks that relate to the management of the object within the programming language. For example, C++ requires explicit memory management through the use of destructor methods. In Java technology, the `equals` and `toString` methods are considered object management methods.
- Unit testing

  You can supply specific methods in your classes to perform unit testing. In Java technology, you could use a `main` method to perform unit tests. Alternatively, you could build separate test classes or use a unit testing framework, such as JUnit.
- Recovery, inverse, and complimentary operations

  Recovery methods provide a means to recover from a complex operation. Such methods are often used by UIs to provide a way to undo an operation. An example of an inverse method and a complimentary method is that if you have a `zoomIn` method, you will probably need a `zoomOut` method and a `zoomByPercentage` method respectively.

The UML syntax for a method declaration is as follows:

```
[visibility] name [({[param] [:type]}*)] [:return-
value] [{constraint}]
```

Suppose that during Robustness analysis, the `ResvService` class needs to support three operations: get a list of properties, create a reservation, and save the reservation. During design, these operations must be transformed into method declarations. Figure 13-17 illustrates this refinement.

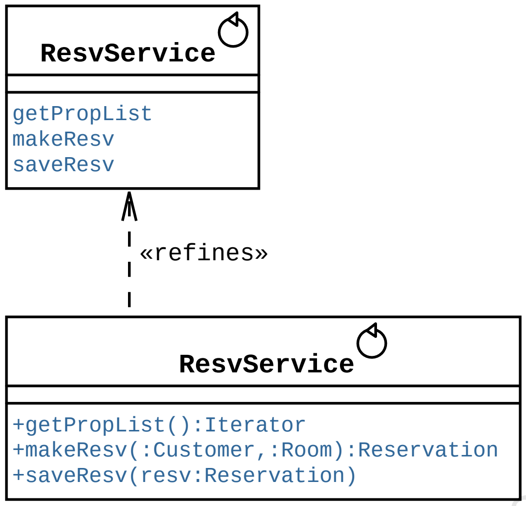
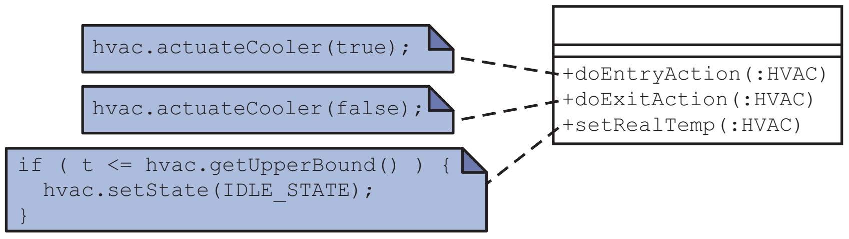


Figure 13-17 Example Refinement of Methods

Annotating Method Behavior

UML annotations can be attached to each method to document the behavior of the method. The behavior may be documented as simple text description, pseudo code, or actual code. Figure 13-18 shows an example of attaching UML annotations to the methods of the `CoolingSate` class.



```
hvac.actuateCooler(true);
```

```
hvac.actuateCooler(false);
```

```
if ( t <= hvac.getUpperBound() ) {
  hvac.setState(IDLE_STATE);
}
```

Figure 13-18 Example of UML Method Annotation

Declaring Constructors

Constructors use similar syntax as methods and are similar to methods, except that there is no return type. Also, in Java technology the name of constructors is the same as the class name. Because constructors look very much like methods, it is good practice to separate the constructors from the methods using the «constructor» stereotype. In many languages, you are provided with a default constructor, which take no arguments. However, if you provide any other constructor, this default constructor is not provided. Some frameworks require you to provide a no-argument constructor even if you do not need the constructor in your code. Constructors can be annotated in the same way as methods. Figure 13-19 illustrates this.

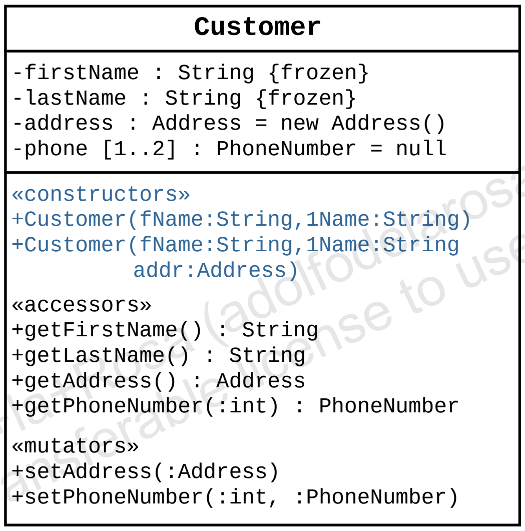

Customer
-firstName : String {frozen}
-lastName : String {frozen}
-address : Address = new Address()
-phone [1..2] : PhoneNumber = null
«constructors»
+Customer(fName:String,lName:String)
+Customer(fName:String,lName:String
addr:Address)
«accessors»
+getFirstName() : String
+getLastName() : String
+getAddress() : Address
+getPhoneNumber(:int) : PhoneNumber
«mutators»
+setAddress(:Address)
+setPhoneNumber(:int, :PhoneNumber)

Figure 13-19 Example Constructors

Reviewing the Coupling and Coherency of your Model

Reviewing Coupling

Ideally, your model should have the lowest coupling while maintaining the FRs and NFRs.

Coupling is a matter of degree. Figure 13-20 shows two versions of a lending library system—one with very high coupling and the other with low coupling. In the Low Coupling version, it would be quick and easy to access the `Book` class from the `Loan` class, but slower and more complex to access the `Loan` class from the `Book` class. In the Very High Coupling version, the access would be quick and easy in both directions. However, the code to add a loan would be more complex and slower as both `Book` and `Loan` will need to be updated. In addition, if adding a loan needs to be performed atomically in a multithreaded system, it would be more complex and might have a negative impact on performance.

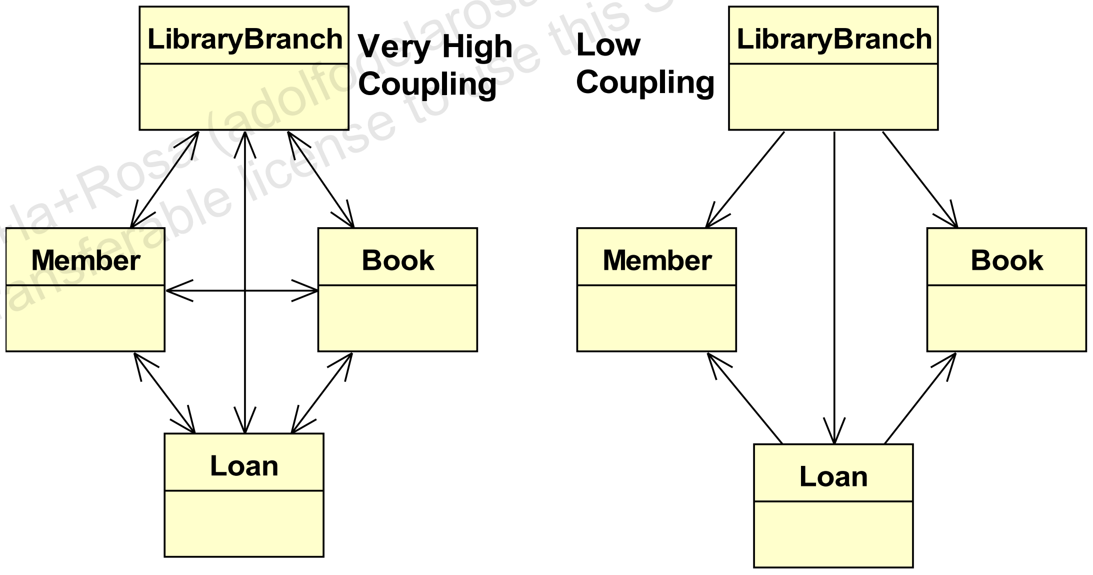


Figure 13-20 Example Comparing High and Low Coupling

Note – The low coupling shown in the figure might not be the lowest coupling possible.

Reviewing Cohesion

Ideally, your model should have highly cohesive components.

Cohesion is a matter of degree; however, more cohesive components tend to be easier to understand, code, test, and adapt to future requirements. Figure 13-21 shows an example of a lending library system. In the Lower Cohesion version, the member and loan details are in the same class, whereas the member and loan details have been separated in the Higher Cohesion version.

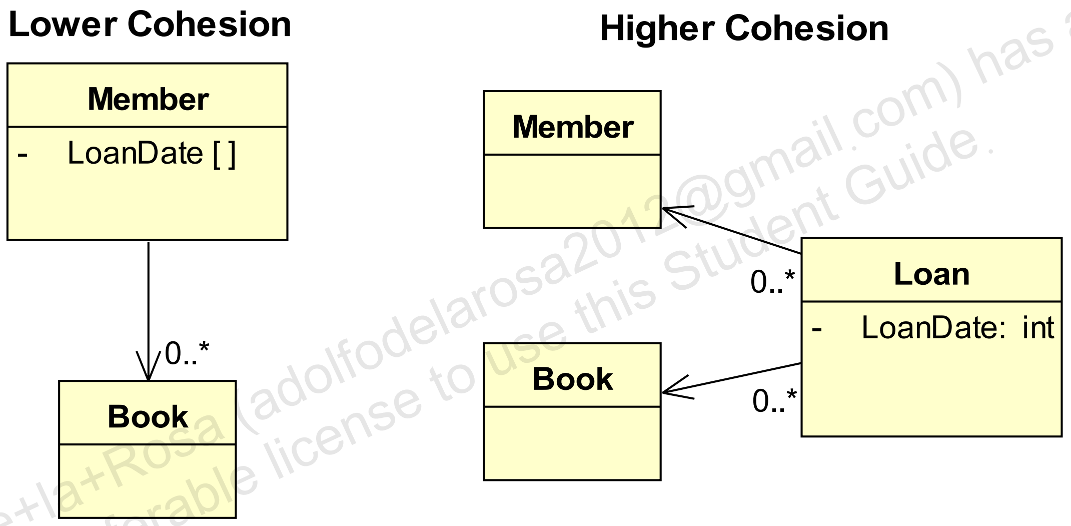


Figure 13-21 Example Comparing High and Low Cohesion

Creating Components with Interfaces

Classes can be grouped into cohesive components with well-defined provided and required interfaces. Figure 13-22 shows an example of a lending library system where all the classes related to membership are grouped into the `Membership` component. The `MemberService` class is split into the `MemberService` interface, which is shown in the diagram as a provided interface, and the `MemberServiceImp` class, which contains the implementation and is hidden inside the component.

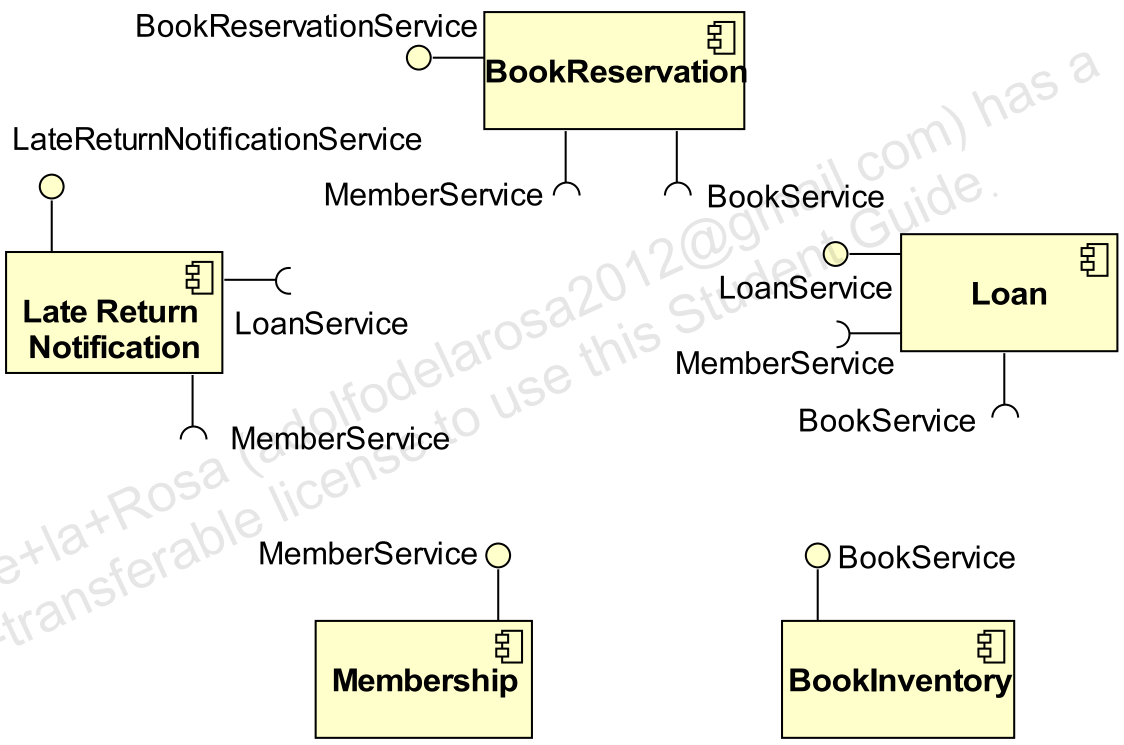


Figure 13-22 Example of Components with Required and Provided Interfaces

You may show dependency arrows in the previous example. However with the required interface notation, these arrows are often redundant. Figure 13-23 shows dependency arrows between the required and provided interfaces.

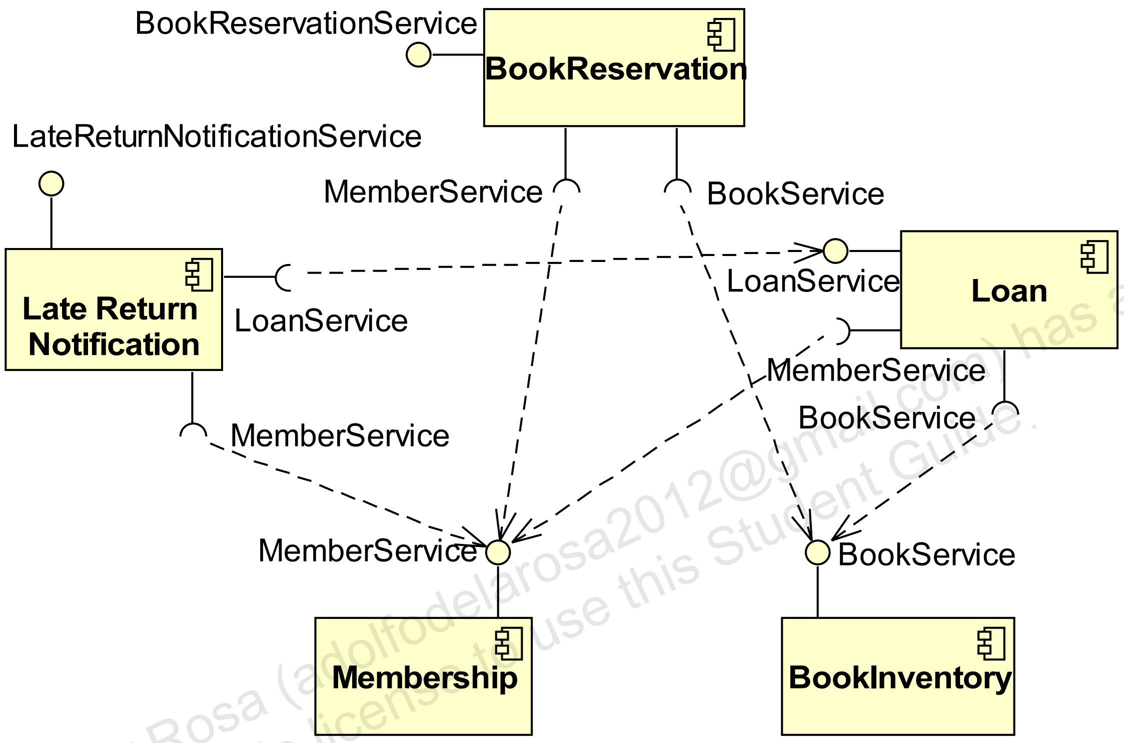


Figure 13-23 Example of Components with Required and Provided Interfaces and Dependency Arrows

Summary

In this module, you were introduced to the process of refining the Domain model. Here are a few important concepts:

- During the Design workflow, you must refine the Domain model to reflect the implementation paradigm.
- This module described how to refine the following Domain model features: attributes, relationships, methods, constructors, and method behavior (by using annotations).
- The classes should be reviewed to ensure that they maintain high cohesion and low coupling.
- Classes can grouped into cohesive components with well-defined interfaces.

Module 14

Overview of Software Development Processes

Objectives

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

- Explain the best practices for OOSD methodologies
- Describe the features of several common methodologies
- Choose a methodology that best suits your project
- Develop an iteration plan

Additional Resources

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

- Arlow, Jim, Ila Neustadt. *UML and the Unified Process*. Reading: Addison Wesley Longman, Inc., 2002.
- Beck, Kent. *eXtreme Programming eXplained*. Reading: Addison Wesley Longman, Inc., 2000.
- Cockburn, Alistair. *Agile Software Development*. Reading: Addison Wesley Longman, Inc., 2001.
- Folwer, Martin. *Refactoring (Improving the Design of Existing Code)*. Reading: Addison Wesley Longman, Inc., 2000.
- Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.

Reviewing Software Methodology

As described in "Describing Software Methodology" on page 2-4, methodology describes the top-level organization of a software development project. This structure usually includes phases, workflows, activities, and artifacts that transform the user requirements into a deployed software system.

Note – There are some disagreements in the industry about terminology. Some people use the term *macro-phase* instead of *phase* and *micro-phase* instead of *workflow.*

This process is illustrated in Figure 14-1.

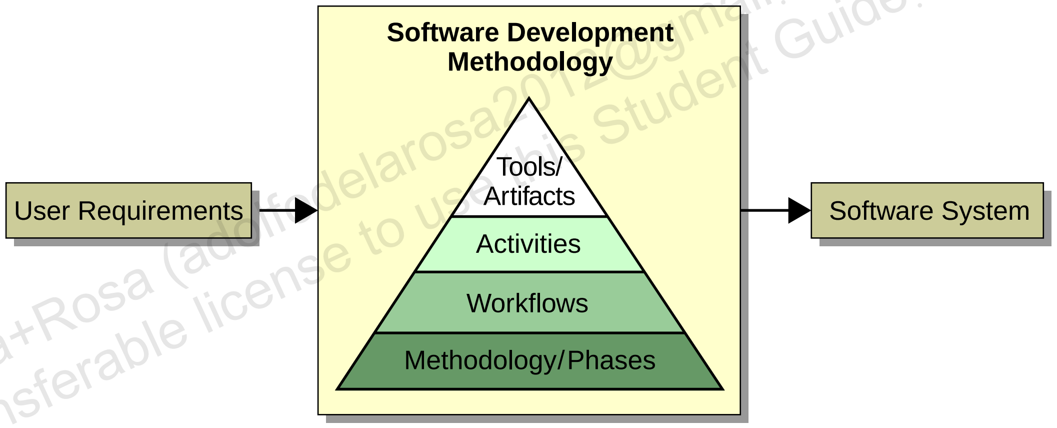


Figure 14-1 Software Development Methodology

This module provides an overview of the best practices in software methodology, an overview of several mainstream and modern methodologies, and an approach on how to decide which methodology is appropriate for your development team.

Exploring Methodology Best Practices

This section provides an overview of the best practices in software methodology. These include:

- Use-case-driven
- Systemic-quality-driven
- Architecture-centric
- Iterative and incremental
- Model-based
- Design best practices

Use-Case-Driven

"A software system is brought into existence to serve its users." (Jacobson USDP page 5)

To know understand how a software system is to be built, you must first understand what functions the system needs to perform for the users of the system. This means that the whole development process is Use-case-driven because use cases capture the functional requirements of the system.

Here are the main issues:

- All software has users (human or machine).

  These are called *actors*. The terms *user* and *actor* are used interchangeably in this course. An actor is some entity outside of the system, such as a human user or an external system (for example, a credit card authorization system).
- Users use software to perform activities or accomplish goals.

  These are called *use cases*. Other methodologies might call this *user stories*, but the idea is the same.
- A software development methodology supports the creation of software that facilitates use cases.
- Use cases drive the design of the system.

The design workflow takes the use cases and creates a set of software components that implement the functionality specified in the use case. The designed Solution model is then used to implement the software system.

Systemic-Quality-Driven

The complete requirements of a software system include functional and non-functional requirements. *Systemic qualities* is the name given to non-functional requirements by SunTone^{SM} Architecture Methodology. A related term is *quality of service*, which refers to the qualitative characteristics of the system.

Examples include:

- Performance – such as responsiveness and latency
- Reliability – the mitigation of component failure
- Scalability – the ability to support additional load, such as more users

Systemic qualities drive the Architecture baseline of the software. After the Architecture baseline has been developed, the software designers can craft the remaining functional requirements into this baseline.

Architecture-Centric

"Architecture is all about capturing the strategic aspects of the high-level structure of a system." (Arlow and Neustadt page 18)

Many areas of risk in software projects involve quality of service issues; such as scalability, availability, and extensibility. The Architecture workflow deals with these issues; therefore, it is a best practice to perform the Architecture workflow as early in the development process as possible. This practice mitigates risk earlier in the process rather than later.

Strategic aspects are:

- Systemic qualities drive the architectural components and patterns.

The Architecture baseline includes all components and architectural patterns that satisfy the NFRs. For example, scalability can be achieved with redundant services distributed across multiple server hosts.

- Use cases must fit into the architecture.

  The Architecture baseline defines the structure into which all of the designed components (those that satisfy the use cases) must fit. For example, in a web application, user interface component might be implemented using servlets and JavaServer Pages™ (JSP™) technology pages using a Model 2 architecture.

High-level structure is:

- Tiers, such as client, business, and resource

  Module 11, "Introducing Architectural Concepts and Diagrams," suggests that a software system be separated into components in several logical tiers. The Client tier includes all user interface components, the Business tier includes all business logic and domain entity components, and the Resource tier manages the persistence of business entities.

- Tier components and their communication protocols

  Components in adjacent tiers must be able to communicate to get their work done. Identifying or creating these protocols is a critical element of the Architecture baseline.

- Layers, such as application, platform, hardware

  In a component-based system, Application layer components are built upon specific APIs. For example, in a Java technology web application, components are built upon the servlet and JSP specifications and APIs.

  Therefore, the Application layer components depend on the APIs and the platform upon which they built. These system layers are another important consideration in the Architecture baseline.

Iterative and Incremental

"Iterative development focuses on growing the system in small, incremental, and planned steps." (Knoernschild page 77)

Requirements change. This is the unwritten law of software development. Except for very rare situations, you must assume that you will not know the complete set of requirements at the start of a software project. Iterative development makes this assumption a built-in feature of the software methodology.

Here are the key ingredients for iterative development:

- Each iteration includes a complete OOSD life cycle, including analysis, design, implementation, and test.

  Within an iteration, you will perform all of the OOSD workflows: requirements gathering, requirements analysis, architecture, design, implementation, testing, and deployment.

  Depending on the stage of development, you might spend more time on analysis and architecture than on implementation and testing. However, at the end of each iteration you should have an integrated and working (albeit incomplete) system.

  Seeing a working system early in the OOSD provides momentum to the development team and gives the client a real indication that progress is being made.
- Models and software are built incrementally over multiple iterations.

  Because you never have a complete set of requirements, you will never have a complete model of the system. That constraint has important implications:

  - First, you should only build models (and the corresponding artifacts) that are required for team communication. Building too many models or diagrams will put high demands on the team to keep them up-to-date as the project moves through successive iterations.
  - Second, the client should have their expectations set appropriately. If the client sees a prototype of the user interface, they might get the impression that more of the underlying system exists when in fact it does not.
- Maintenance is simply another iteration (or series of iterations).

  As mentioned earlier, maintenance is not a workflow, but rather a new revision of the system. Simple maintenance (such as bug fix releases) can be accomplished in a single iteration. More complex maintenance (requiring significant enhancements) might require multiple iterations.

Model-Based

Models are the primary means of communication between all stakeholders in the software project. Artifacts are physical representations of our mental models. Modeling is a fundamental property of how humans think. Whether explicit or not, all humans have mental models of reality and the systems that they develop. The choice to give these models physical representation through an artifact is a decision of the project manager.

There are several types of model artifacts:

- Textual documents

  Requirements specifications and use case scenarios require both formal and informal text. Documents, such as the Vision and System Requirements Specification, are examples of requirements-level artifacts.

- UML diagrams

  The UML provides a rich set of diagrams to represent visual views of your mental models of the proposed software system.

- Prototypes

  Prototypes are small-scale software systems built to model a certain aspect of the proposed software system. There are two main types of prototypes: technology and user interface. Technology prototypes explore a new, unknown, or risky area of technology. User interface prototypes explore the visual representation of the proposed system and enables the client and UI designer to explore various usability concerns.

Models can server many purposes:

- Communication

  Model artifacts communicate your mental models to other project stakeholders.

- Problem solving

  Some models can be constructed (either singularly or in a team) to solve a particularly difficult problem. These models tend to be *throw-away* because they serve a short-term need.

- Proof-of-concept

Some models (especially prototypes) are created to demonstrate the feasibility of a concept. These also tend to be throw away in nature. If proof-of-concept prototypes are shown to the client, it is important to set the expectation correctly that this prototype will *not* be part of the final system.

Note – Some methodologies (such as XP) do not put a high value on creating artifacts of models. However, XP's emphasis on close ties between users and developers reduces the risk of not understanding the use cases (and other requirements) of the system.

Design Best Practices

Understanding and applying design-level best practices can improve the flexibility and extensibility of a software solution. You will be introduced to these best practices throughout the course.

These best practices include the following:

- Design principles

  Design principles are rules about how software components (including classes and packages) interact or are combined to provide flexibility, extensibility, and decoupling. These principles provide the rules from which patterns arise.

  Example design principles include: Separation of Concerns, Dependency Inversion Principle, and Stable Dependency Principle. These are described in other modules of the course.
- Software patterns

  A software pattern is a *repeatable solution to a reoccurring problem within a context*. Patterns describe expert solutions to everyday problems.

  Module 10, "Applying Design Patterns to the Design Model," provides more information about design patterns.
- Refactoring

  Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure." (Fowler page xvi)

  Refactoring is done all of the time during software development and especially during maintenance. Refactoring is recognized as a legitimate and significant practice. Programmers should be encouraged to develop the skill of refactoring. Fowler's seminal book, *Refactoring*, provides a detailed exploration of this topic.
- Sun Blueprints

  Sun Microsystems publishes documents that bring together best practices in a wide variety of technical areas including J2EE development, cluster configurations, and so on. For more information visit the web page at `http://www.sun.com/blueprints/`.

Surveying Several Methodologies

In this section you will be introduced to several mainstream and new OOSD methodologies:

- Waterfall
- Unified Software Development Process (USDP or just UP)
- Rational Unified Process (RUP)
- Scrum
- eXtreme Programming (XP)

Waterfall

The Waterfall methodology is one of the oldest development processes. It is presented here as a *baseline* to evaluate other methodologies. The Waterfall methodology has the following characteristics:

- Waterfall uses a single phase in which all workflows proceed in a linear fashion.

  Requirements gathering precedes analysis, which precedes architecture, and so on. Variations of the Waterfall methodology permit backing up to the previous workflow.
- This methodology does not support iterative development.

  The Waterfall methodology proceeds as a single, large iteration.
- This methodology works best for a project in which all requirements are known at the start of the project and requirements are not likely to change.

  This is the fundamental flaw of the Waterfall methodology because it is very unlikely that the system requirements can be fully specified at the beginning of the development process. It is even more unlikely that requirements changes will not occur during development.
- Some government contracts might require this type of methodology.
- Some consulting firms use this methodology in which each workflow is contracted with a fixed-price bid.

  Some consulting firms are required (by the client) to produce a system on a fixed-price bid. Changes to requirements can be managed by the use of an “engineering change order” agreement between the consultant and the client.

The Waterfall methodology is illustrated in Figure 14-2.

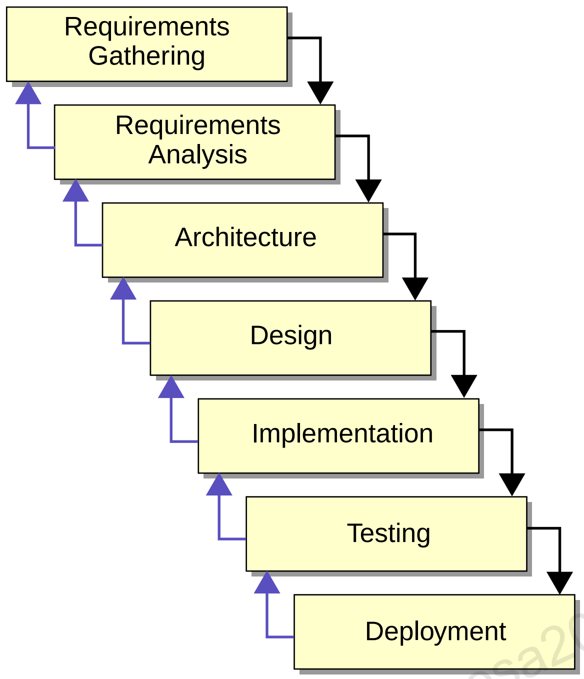


Figure 14-2 Waterfall Methodology

Note – Figure 14-2 shows arrows pointing up to a previous workflow. This feature might not be considered part of a formal definition of the Waterfall methodology.

Unified Software Development Process

The Unified Software Development Process (USDP) is the non-proprietary version of Rational's methodology created by Grady Booch, Ivar Jacobson, and James Rumbaugh. The USDP is also called the Unified Process (UP).

The UP methodology organizes the development process into four phases:

- *Inception* – Creates a vision of the software

  This phase focuses on understanding the business case for the proposed system. Also, a preliminary architecture is proposed, important project risks are identified, and the elaboration phase is planned in detail.
- *Elaboration* – Most use cases are defined plus the system architecture

  The purpose of this phase is to reduce project risk by creating an architecture baseline upon which the rest of the software system will be constructed. At the end of this phase, the project manager should have enough information to plan the construction and transition phases.
- *Construction* – The software is built

  Through multiple iterations the software system is built incrementally by adding additional use cases to the architecture baseline. At the end of this phase, the system is ready for Beta release.
- *Transition* – Software moves from Beta to production

  This phase prepares the system for production release. Activities include acceptance testing, debugging, training, and building the production environment.

The UP methodology is illustrated in Figure 14-3. What is not shown in this diagram is that multiple iterations can be constructed in each phase.

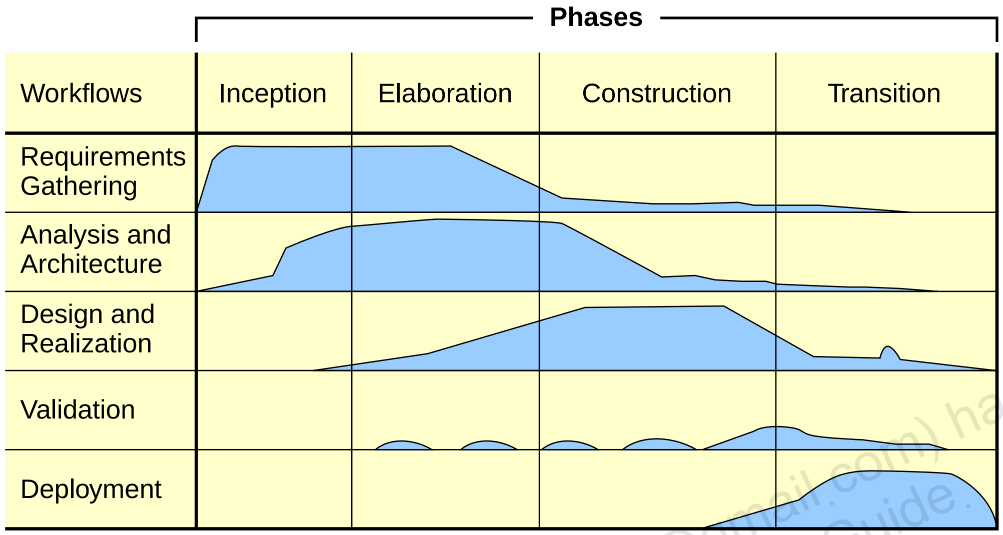


Figure 14-3 UP Methodology

The UP methodology embraces all of the best practices listed in the previous section, with less focus on systemic-qualities-driven. The UP is also focuses on using UML to develop modeling artifacts and on planning the activities of the development team.

Rational Unified Process

RUP is the commercial version of the UP methodology created by Grady Booch, Ivar Jacobson, and James Rumbaugh. RUP is UP with the support of Rational's tool set. These tools manage the phases, workflows, and artifacts throughout the project life cycle. The set of tools that support RUP are shown in Figure 14-4.

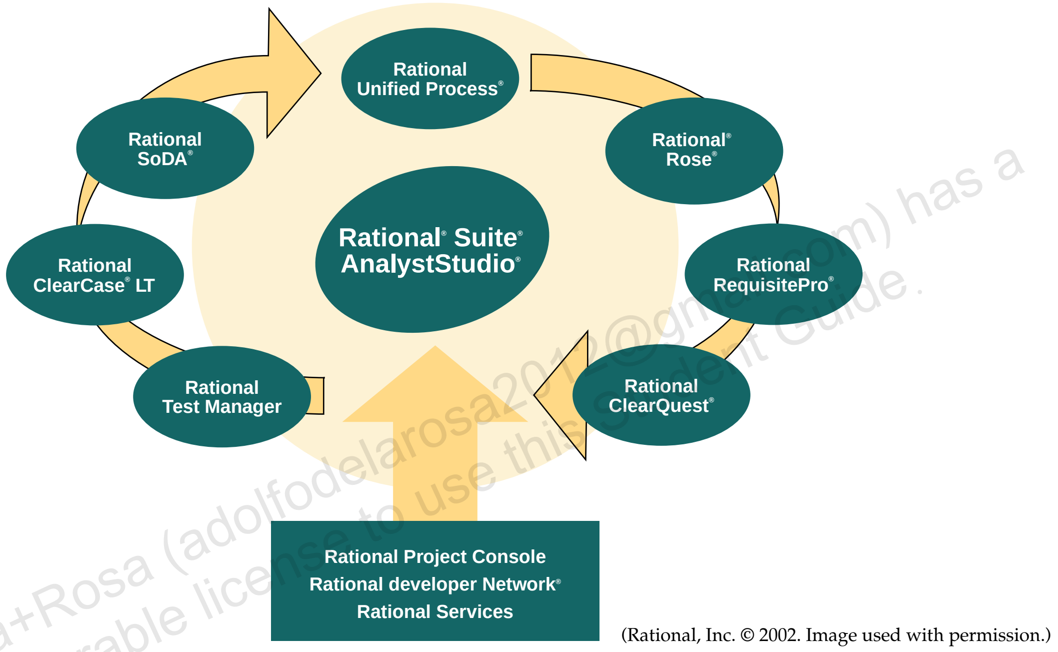


Figure 14-4 Rational's Tool Set for the RUP Methodology

Scrum

Scrum is an iterative and incremental development framework that is often used in conjunction with agile software development.

The Scrum framework includes a set of practices and roles. The key features of Scrum are:

- Each Sprint is typically 15 to 30 days and produces a deliverable increment of Software.
- A subset of features or requirements are moved from the Product backlog to the Sprint backlog at the beginning of each Sprint.

  The Product backlog is a high-level document containing system requirements, estimates of priorities, and the required development effort.

- The requirements in the Sprint backlog are developed during the Sprint.

  Each Sprint produces a version of the software that the client can use.
- The Sprint progress is reviewed every 24 hours in a daily Scrum meeting.
- The Scrum framework requires team-oriented responsibilities.

Figure 14-6 shows the Scrum development methodology.

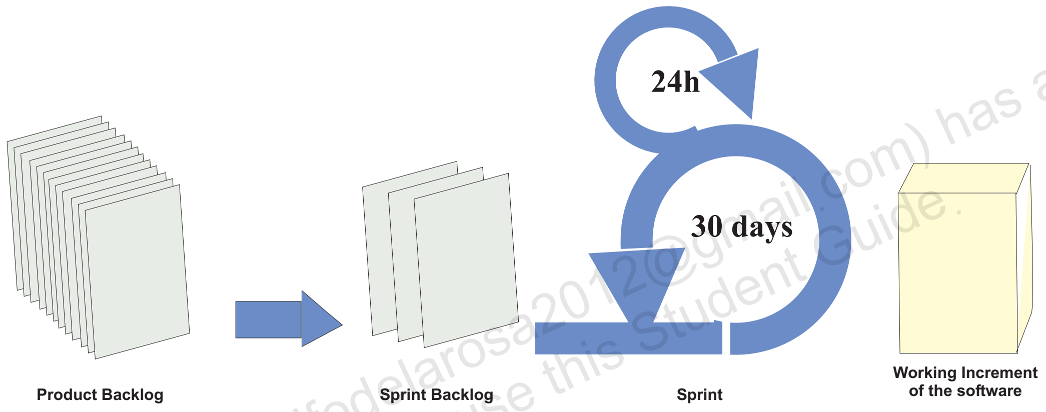


Figure 14-5 Scrum Development Framework

eXtreme Programming

> "XP nominates coding as the key activity throughout a software project." [Erich Gamma, forward to Beck's XP book]

XP is one of several relatively new methodologies, called *Agile methodologies*. In many ways, XP is a dramatic departure from traditional methodologies. XP focuses on facilitation of change and communication within the team and with the clients.

It is impossible to give it a rich presentation in this module. However, here are a *few* key ideas:

- Pair programming – If code reviews are good, then review code all the time.

  Developers are paired together during coding sessions. As one person types the other can be reviewing the code and making suggestions to clean up or refactor the code.

- Testing – If testing is good, then test all the time, even the customers.

  Testing is the cornerstone of XP. No system code should be written until a test is developed to verify that the system code is written correctly. Creating the tests helps the developers think through the code solution before they write the code.

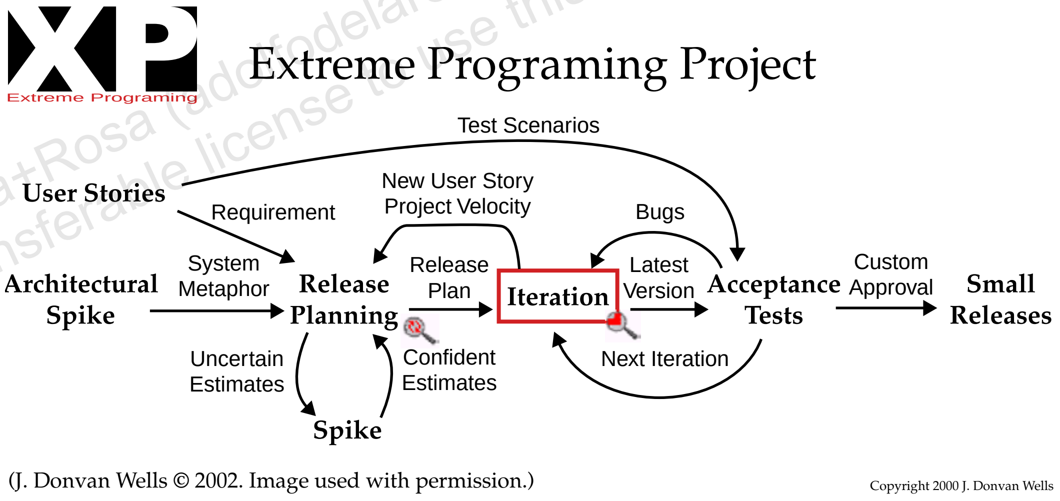
- Refactoring – If design is good, then make it part of everybody's daily business.

  Pair programming encourages continual refactoring. Refactoring code is essentially doing design while you code.

- Simplicity – If simplicity is good, then always leave the system with the simplest design.

  This idea is powerful: The system should always have the simplest design to support the use cases; no more, no less.

Iterative development is essential and is taken to the extreme with XP. Small, almost daily, releases are built from *user stories* which are also called use case scenarios. This methodology is illustrated in Figure 14-6.



(J. Donvan Wells © 2002. Image used with permission.)

Figure 14-6 XP Methodology

Choosing a Methodology

Among the numerous factors that can guide the choice of a methodology for a given product, this section will focus on the following factors:

- Company culture – Process-oriented or product-oriented
- Make up of team – Less experienced developers might need more structure and people have distinct job roles
- Size of project – A larger project might need more documentation (communication between stakeholders)
- Stability of requirements – How often requirements change

Choosing Waterfall

When to use:

- Large teams with distinct role

  Large teams require a great deal of structure to support the communication problems between developers and subteams. Sometimes, teams exist only for a single workflow. For example, a team of business analysts might perform a complete requirements analysis. They then hand off their work to the architecture team, and so on.
- Choose waterfall when the project is not risky

  This is the most important factor in deciding to use Waterfall. A project is not risky when the domain of the requirements and the technologies to be used in the proposed system are well known.

Issues:

- Not resilient to requirements changes

  It is difficult to go back to a previous workflow to make changes to the system. It is usually acceptable to go back to the last workflow, but it is nearly impossible to go from the Implementation workflow back to the Requirements Gathering workflow.
- Tends to be documentation heavy

  The Waterfall methodology depends on well-defined documentation deliveries between workflows. These documents tend to be large from the start because everything in the current workflow must be defined before proceeding to the next workflow.

Choosing UP

When to use:

- Company culture is process-oriented

  There is a continuum along the lines of *process orientation* in which on one side there is too much process (large amount of documentation and formal ceremony) and too little process (what some might characterize as *hacking*). UP tends to favor a process-oriented culture.

- Teams with members that have flexible job roles

  While UP encourages the delineation of job roles, it is not as strict about the separation of workflows and the distinction of job roles that perform each workflow, which is the hallmark of the Waterfall methodology.

- Medium- to large-scale projects

  Projects of a certain size require documentation so that all of the stakeholders of the project can understand the problem and solution domains clearly. UP also supports documentation used to communicate across workflow boundaries.

- Requirements are allowed to change

  Because UP uses an iterative process, requirements changes can be folded into later iterations. Every iteration goes through the complete set of workflows, so new requirements can be supported in another iteration.

Issues:

- Tends to be process and documentation heavy

  UP makes extensive use of the UML to diagram various models developed in the workflows. This close tie to the UML makes UP documentation heavy.

- This is overkill for small projects

  UP is overkill for small-scale projects because UP requires more process and modeling than a small project needs.

Choosing RUP

When to use:

- Same reasons as UP

Because RUP is based on UP, it is applicable whenever UP is applicable.

- Your company owns Rational's tool set

  RUP requires the Rational tools. If your company already uses Rational tools, then you are in a position to use the RUP methodology.

Issues:

- Same issues as UP

  Because RUP is based on UP, it has the same issues as UP.

- Tool set learning curve

  There are many tools used by RUP. There is a steep learning curve to using these tools properly within the RUP methodology.

- Tools lock the team into a process

  The tool set will lock you into the RUP methodology. However, the RUP tools can be customized. The customization process is difficult.

Choosing Scrum

When to use:

- Priorities of the requirements are constantly changing

  Features to be developed in each Sprint are decided at the beginning of the Sprint and not at the beginning of the project.

- When you need to deliver a working increment of the software every 30 days

  You can deliver a working incremental release of the software at the end of each iteration.

Issues:

- Requires a committed team

  Each Sprint team is responsible for the delivery of that Sprint increment. Scrum refers to their role as Pigs, as their bacon is on the line.

- Primary focus is on the functional requirements for each Sprint

Non-functional requirements might be not be considered adequately. The optimum choice of system architecture is primarily dependent on non-functional requirements. Therefore, a Scrum process might result in a non-optimum system architecture development.

Choosing XP

When to use:

- Company culture permits experimentation

  The XP methodology is a radical departure from traditional development processes. It must be part of the company culture to permit experimentation with the process. If your company wants to experiment with the XP methodology, then a small project that does not have significant schedule constraints would be a good candidate.

- Small, close (proximity) teams with flexible work spaces

  Pair programming and flexible *open* work spaces are fundamental to the XP development process. A project using XP is likely to fail if the teams are distributed geographically (even within the same building).

- Team must have as many experienced developers as inexperienced

  To facilitate pair programming and coaching, the development team must have a good ratio of experienced to inexperienced developers. A team full of inexperienced developers will likely fail using XP.

- Requirements change frequently

  One of the greatest benefits of XP is that the process easily supports changing requirements.

Issues:

- Tends to be documentation light

  User stories and the code are documented by using the XP methodology. XP discourages creating models. Design is done in real-time and documented in the code.

Project Constraints and Risks

Project constraints and risks are often managed by Project Architect.

Project constraints and risks should be assessed during the initial stages of the development process.

Project Constraints

Project development constraints are often confused with NFRs, which are the runtime constraints of the system.

Project constraints primarily focus on constraints during the development process.

Typical project constraints are:

- Project must be developed on a specific platform

  For example, the application and the web server must run on Sun hardware.
- Project requires specific technologies

  For example, the application server must run on Java Platform, Enterprise Edition (Java EE), the Web Services service must use SOAP (Simple Object Access Protocol). The data store must use Oracle RDBMS (Relational Database Management System).
- Project has a fixed deadline

  For example, the project must be ready by the next olympic games.

Note – A detailed discussion of the issues and risks around making a project calendar-constrained are beyond the scope of this course. For more information, read Fred Brooks *The Mythical Man-Month*.

- System interacts with specific external systems

  Integration with specific external systems is an important constraint on the project.

Project Risks

Every project involves some level of risk. Any situation or factor that could lead to an unsuccessful conclusion to a project defines a risk. Each constraint will usually affect several risks factors, some positively and some negatively.

The five main risk areas are:

- Political
- Technological
- Resources
- Skills
- Requirements

Political Risks

A political risk is any situation in which the project is competing with other projects (internal or external) or in which the project might conflict with a governmental law. These are extremely sensitive areas to discuss with the business owner. The best thing to do is to listen closely for warning signs in what the business owner is saying.

Here are a few warning signs:

- When there is a competing project or competitor.

  Sometimes the competition comes from within. In large companies, there might be two or three groups trying to solve the same problem. Usually this situation might not be known.

  Other times the competition comes from a competing company with a similar product. The client company might choose to pursue the project, but if the competitor has a better product or gets to market earlier, then the client company might terminate project.
- When the project manager's boss has revoked funding, equipment, or human resources without warning in the past.

  Financial situations might cancel some projects. Try to ascertain if such issues have cancelled other projects within this organization.
- When there are interpersonal problems or infighting either within the development team or within the management hierarchy.

Some large companies have organizations that are not mutually cooperative. Does the project depend on help from another organization that is hostile to the business owner's organization?

Sometimes interpersonal problems occur within the project manager's organization itself. Are there members of the development team that have had fights over design strategies or architectural decisions? How well were these issues resolved? Were they resolved at all?

- When the project conflicts with governmental laws and regulations.

  Beyond interoffice political risks, there could also be political risks with regard to governmental regulations. For example, if your product is produced in the United States but sold abroad and it uses cipher technologies, then you will have to deal with export considerations.

Technology Risks

A technology risk exists when the project might use a technology that is unproven, cutting edge, or difficult to use. Some of the technologies that you might encounter include: web applications, web services, hand-held devices, voice recognition systems, telephone integration systems, specific computer languages, computer platforms, and so on.

Sometimes even the software development methodology might be a technology risk. For example, if the business owner wants the development team to use XP and the company has never tried XP before.

Here are a few warning signs:

- The business owner uses a lot of technology buzz words, and does not really understand their meaning.

  This is a dangerous situation that needs to be dealt with early. Try to steer the business owner away from any discussion about specific technologies and back toward what the system must do. If you have a good rapport with the business owner, you might attempt to explain some capabilities these technologies and how they might be useful to the project.
- The business owner insists that the project take a specific technological direction.

  If the business owner persists, then make note of the specific technology. You can have a discussion about risks in follow-on interviews.
- The business owner wants to use cutting-edge technologies to solve the business problem.

  This situation is also dangerous. The business owner might be very knowledgeable about certain technologies and the recommendation to use a cutting edge technology might be appropriate. However, it is still a risk if the development team has never worked with that technology.
- The new system must integrate with a legacy system.

  Integrating to a legacy system is almost always risky because often the original architect or development team for the legacy system has left the company. This risk requires a great deal of investigation to understand how the legacy system works and what would be involved to enable the new system to communicate with the legacy system.

Resource Risks

A resource risk exists when the project does not have all of the resources (human, equipment, or money) required for successful completion.

Here are a few warning signs:

- The business owner mentions that the project is under a tight budget.

  Budget constraints can be just as difficult to mitigate as schedule constraints because on most projects time equals money. You should provide the client with realistic schedule and cost information. It is then the business owner's decision to proceed with the project or not. Often what happens is that there is a negotiation about what is included in the first and follow-on releases of the system.

  It is important to correctly set expectations with the client. That is exactly why you will ask the business owner to rate each use case as: essential, high-value, and follow-on.

- The IT staff is currently over committed.

  Many organizations with IT departments are over committed; that is, all of the developers are working on other projects. If you are a consultant, then this is only an issue if you are required to have several client IT developers on your team. Moving client IT developers to your team is done to encourage technology transfer from the consulting organization to the client's IT organization.

- The project is calendar-driven.

  This risk is roughly equivalent to the first issue: tight budget.

Skills Risks

A skills risk exists when the project team does not have the necessary knowledge or experience.

Here are a few warning signs:

- When the project is constrained to use a specific technology but the development team has not been trained in that area.

  For example, if the project requires a web application and no one on the team has ever built one.
- When the project will be developed in a language that is not known by the development team.

  This is especially problematic when there is a language *paradigm shift* such as from structured programming to object-oriented programming.

Requirement Risks

A requirement risk exists when a given use case, scenario, or NFR, is not completely known.

Here are a few warning signs:

- When business owner says something like "I will know it when I see it."

  Sometimes it is hard for the business owner to describe in words how a use case functions. In this situation, try to have the business owner show you how it works or draw you a diagram of how it works. Do not spend too much time on this; you can always record this as a risk and move on.
- When the business owner cannot provide a scenario for the use case.

  Sometimes it is hard for the business owner to know how a use case functions. Some use cases require detailed interviews with a domain expert. It is sufficient for the Vision document to simply record this as a risk and to reevaluate the use case with an expert. Make sure that you ask the business owner who is the expert in that area.
- When the business owner does not know that a use case exists.

  This situation is dangerous and might occur if the company is going through several changes or is being merged with another company.

Producing an Iteration Plan

Iteration plans are mainly applicable to iterative and incremental development processes. However, some of the techniques in this topic, can be used to determine the linear order of development in a non-iterative process.

Iteration plans determine the order in which the software components are developed.

If you are using a use-case-driven methodology or any other methodology, consider each use case based on priority, risk, and dependencies.

The iteration plan might be documented in detail early in the project and rigidly followed, or might be loosely planned and open for change as the project progresses.

Prioritizing Use Cases

MuSCoW

Use cases (or user stories, high-level requirements, and product features) are often categorized based on priority. The terms High, Medium, and Low can be used but are open to interpretation. A less ambiguous method that is commonly used is the MuSCoW prioritization technique, where priorities are defined as:

- **Mu**st have this

  Might also be defined as a minimum usable subset.
- **S**hould have this, if at all possible

  Important as Must, but are less time critical or have a workaround
- **Co**uld have this, if it does not affect anything else

  Nice to have. For example, they improve the user experience of the product.
- **W**on't have this time, but would like to include in the future

Assessing Use Case Risk

Consider each use case based on risk. The risk factor considerations should include:

- Complexity of the use case

  A complex use case is usually more risky than a simple use case because of the number of unknown issues that you might encounter during development. However, if a use case is complex, but is similar to the previous work done by the team, the risk factor is reduced.
- Interfaces to external devices and system

  Interfacing to external devices and system often results in unexpected issues. Even the best documented interfaces to external system might miss some critical information.

Architectural Significance

Use cases are architecturally significant if they have project-critical non-functional requirements (NFRs). These NFRs determine the success or failure of the project.

Project-critical NFRs require the architect to choose an architecture that satisfies those NFRs. For example, performance and scalability NFRs can affect the distribution of major components, and the number and type of each server.

Architectural Baseline

An architectural baseline is a subset of system requirements. This baseline can be realized in code early in the project and tested to prove that the chosen architecture satisfies major NFRs.

The baseline code includes the architecturally significant use cases. This process of testing these architecturally significant use cases is often called "Architectural Proof-of-Concept."

In the Unified Process (UP), the architectural baseline should be completed at the end of the elaboration phase.

Timeboxing

Each iteration in an iterative and incremental development process has a fixed time duration. This process is called timeboxing.

The key features of timeboxing are:

- The iteration must complete on schedule.

  Even if the chosen set of use cases is not completely developed, the iteration completes on schedule.
- Unfinished use cases are rescheduled to the next iteration.

  If you have a significant number of deferred use cases, or they are deemed to be more complex than estimated, then you might need to add a new iteration into the schedule, that would extend the project.

Use cases are prioritized. So continuous rescheduling of a specific use case—often called skidding—might indicate that a high risk use case is not developed until almost the end of the project. This rescheduled use case might prove difficult or impossible to build. Therefore, you should avoid skidding.

80/20 Rule

In an iterative and incremental development process, you do not need to fully understand every aspect before moving to the next step.

The 80/20 rule can be applied. For example:

- For 20% of the effort, you can understand 80% of the detail.
- For 20% of the effort, your specification can achieve 80% accuracy.

Missing details and accuracy are found in the subsequent iterations for a fraction of the effort.

This rule works only if the software that you build is flexible. The OO paradigm when applied correctly is flexible with changes being predominantly additive or subtractive.

Producing an Iteration Plan

An iteration plan contains use cases that you intend to develop in each iteration.

In UP, this plan partly occurs during the inception phase and is completely defined by the end of elaboration.

The assessment criteria includes the following items:

- Use case priority
- Use case risk
- Architectural significance
- Estimated time to develop a use case
- Dependency on other use cases

  You can build a simulation of the use case or classes to allow building a dependent use case.

Figure 14-7 shows a subset of the use cases for a library system. There are at least 40 use cases even for the simplest library system.

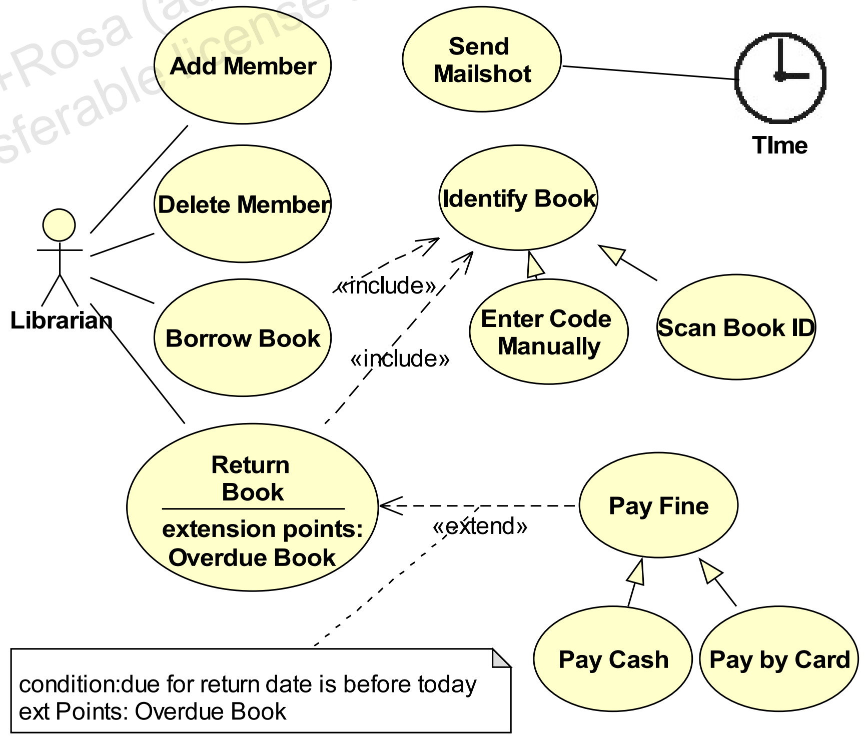


Figure 14-7 Sample Library Use Case Diagram

Table 14-1 shows an example of the information that you might use to determine the iteration plan. In practice, you might not need to build a table as all the information should exist in each Use Case form and Supplementary Specification Document.

Table 14-1 Sample Iteration Plan Details

Use Case Name	Priority	Risk	Architectural Significance	Dependency
Return Book	Must have	Medium	Must be complete within 60 seconds, and easy to use	Borrow Book
Pay Fine	Should have	Medium	None	Return Book
Pay Fine by Cash	Should have	Low	None	Pay Fine
Pay Fine by Card	Could have	High	Transaction with external system must be complete in 30 seconds	Pay Fine
Borrow Book	Must have	Medium	Must be complete within 60 seconds, and easy to use	Add Book Add Member, Identify Book
Identify Book	Must have	Medium	Must complete within 5 seconds	Add Book
Enter Code Manually	Must have	Low	Must be fast	Identify Book
Scan Book ID	Should have	Medium	Interaction with a bar code scanner	Identify Book
Delete Member	Should have	Low	None	Add Member
Send Mailshot	Could have	Low	None	Add Member

Note – The entries in Table 14-1 are highly subjective, and often based on the writer's previous experiences. Therefore, you might disagree with some of the entries. In practice, obtain a consensus of option from all the stakeholders.

Table 14-2 shows a suggested example iteration plan based on the details shown in Table 14-1. This example uses the Add Book and Add Member Book simulations to concentrate on the Borrow Book and Return Book use cases. The first iteration is ignored, which is the inception phase in UP. However, you can build a few use cases in inception to prove the viability of the project. The subsequent iterations of the construction and transition phases are ignored.

Table 14-2 Sample Iteration Plan

Elaboration Iteration 2	Elaboration Iteration 3	Construction Iteration 4	Construction Iteration 5	Construction Iteration 6
Borrow Book	Return Book	Add Book	Add member	Send Mailshot
Identify Book	Pay Fine by Card	Pay Fine by Cash	Modify Member	Delete Member
Enter Code Manually	Scan Book ID			
Simulation of Add Member	Pay Fine			
Simulation of Add Book				

Summary

In this module, you were introduced to several OOSD methodologies. Here are a few important concepts:

- Iterative and incremental development
- Architecture-centric development
- Project risks and constraints
- Developing an iteration plan based on a use case's:
  - Priority
  - Risk
  - Architectural significance

No one methodology fits every organization or project. You can create your own methodology by adapting an existing methodology to suit your requirements and by following the best practices.

Module 15

Overview of Frameworks

Objectives

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

- Define a framework
- Describe the advantages and disadvantages of using frameworks
- Identify several common frameworks
- Understand the concept of creating your own business domain frameworks

Additional Resources

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

- Fowler, Martin. *Analysis Patterns*. Addison Wesley Longman, Inc., 1997.
- Larman, Craig. *Applying UML and Patterns (3rd ed)*. Upper Saddle River: Prentice Hall, 2005.

Description of Frameworks

A software framework is a re-usable software infrastructure that can be extended and configured to provide a specific software solution. The infrastructure can include components, APIs, scripts, support applications, configuration files.

A software framework provides extension points in the framework where the application programmers may make additions or modifications for specific functional requirements.

A framework can provide the infrastructure for:

- One or more tiers such as web presentation, business services, entities, and integration tiers

  These infrastructure files will often facilitate the non-business oriented tasks required by a software system such as transactions, security, or object life-cycle management.
- A specific business domain, for example, insurance, banking or oil exploration

  These infrastructure files are created and packaged but are usually used internally within an organization
- A shared business domain requirement such as resource allocation, event management or billing

  These infrastructure files can be packaged and reused within an organization or sold to third parties

Customization of a framework is done by:

- Extending framework classes or implementing framework interfaces. You implement the framework interfaces or extend the framework abstract classes and provide the specific method implementation required by the framework. In this customization process:
  - Your classes may be less coherent due to their mixed responsibilities
  - Your classes may be more difficult to test outside of the framework
- Informing the framework of the Plain Old Java (POJO) classes it must manage by using configuration files or annotations. POJOs do not extend or implement the framework classes or interfaces, but are

configured using configuration files, or annotations added to your source POJO files. POJOs often allow the framework to be replaced with minimal effort. In this customization process:

- POJOs tend to be more coherent
- POJOs can be easier to test outside of the framework environment

Using Existing Frameworks

List of common frameworks:

- Ruby on Rails – A free software application framework for the Ruby Platform using model-view-controller (MVC) architecture, for rapid web development.
- Spring Framework – An open source application framework for the Java platform and.NET Framework, primarily used for the business tiers, but it has extensions for building the web presentation tier. It is often used in conjunction with hibernate and as an alternative to a JEE application server.
- Java Server Faces (JSF) – An open source web application framework for the Java platform, which uses component based and event-driven approach for rapid application development.
- Hibernate – An open source Java-based framework to perform Object/Relational mapping and queries. This framework fits into the Integration tier.
- Struts – A free open source framework for creating web applications, based on JavaServer Pages (JSP). It helps developers achieve separation of concerns of the model, view and controller, by utilizing the MVC architecture.
- Microsoft .NET – A software framework, designed to run on Microsoft Windows operating systems. It provides a framework for developing in multiple tiers including the web, business, integration tiers.
- Struts 2 – An open source framework for web application development, with improved features over Struts. The improvements include POJO forms and action classes.

Advantages and Disadvantages of Using Frameworks

Following is the list of advantages for using frameworks during application development:

- Developers can focus on the new business problem and need not worry about the infrastructure problems, or common business aspects of the overall application.
- Frameworks usually include good OO practices and patterns.

- Once you have gained experience with a framework, code is easier to write and support

Some of the disadvantages for using frameworks are:

- Your code may become bloated due to the one-size-fits-all approach of the framework.
- Frameworks may be difficult to learn.
- You are restricted by the infrastructure code and cannot usually modify the infrastructure files.
- Changing to an alternative framework may be difficult.

Building Frameworks

A generic framework can be built for a specific business domain. Generic attributes and methods could be specified in the framework. The generic classes could have associations with other generic classes and link attribute methods could be specified in the framework classes.

An insurance company could create a generic insurance framework that would support any insurance domain product. For example:

- Pet insurance
- Car insurance
- Life insurance
- Property insurance
- Public liability insurance

The following examples show two alternative approaches to build a generic framework for an insurance domain with a specialization of pet insurance:

- Figure 15-1 shows a framework based on abstract classes
- Figure 15-2 shows a framework based on interfaces and abstract classes

There are other possible approaches to build frameworks.

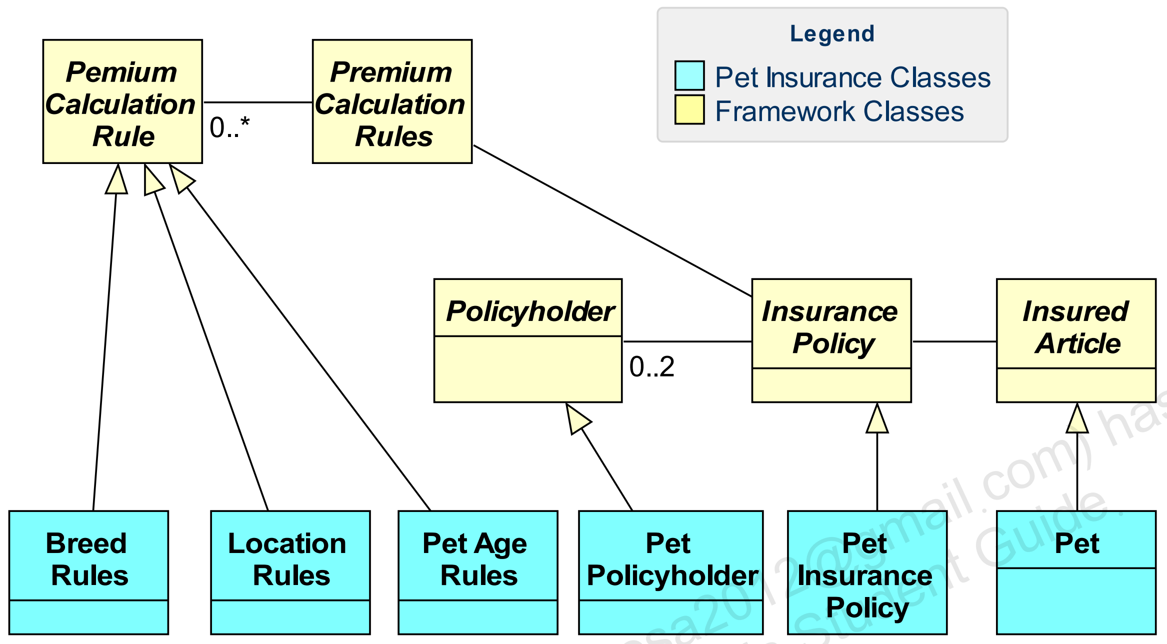


Figure 15-1 Example showing framework of abstract classes

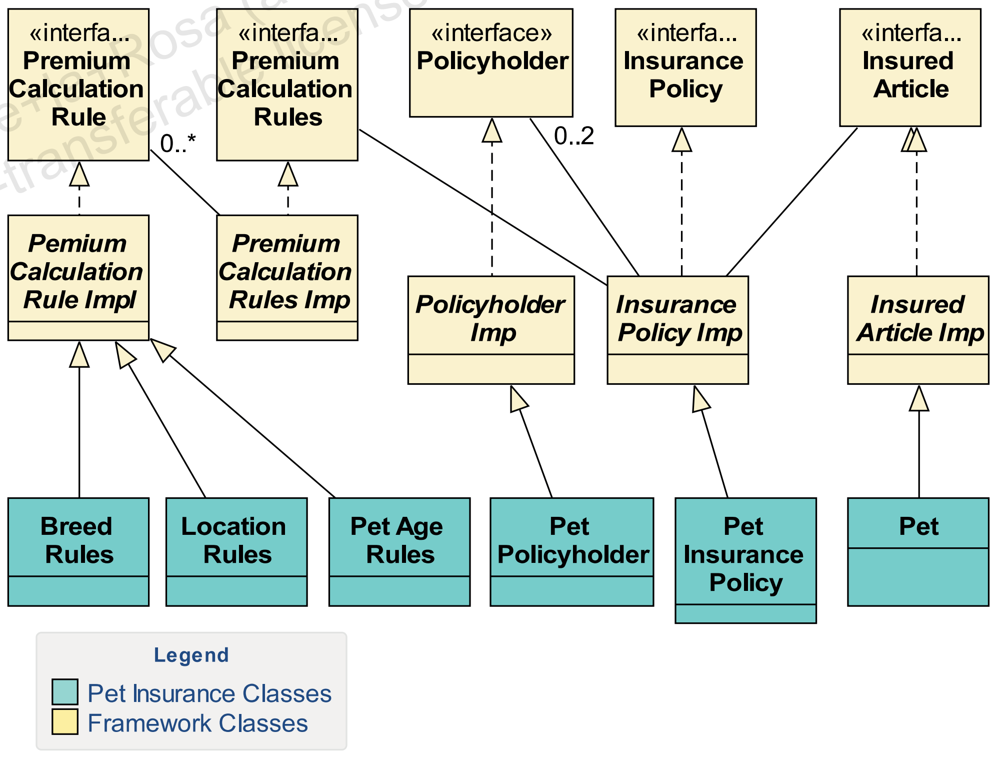


Figure 15-2 Example showing a framework of interfaces and abstract classes

Domain Neutral Frameworks

Domain neutral frameworks are often used for subsystems. These frameworks contain common features required by a variety of different domains. For example:

- A used car sales system could use a trading framework and a billing framework
- A human resource system could use a resource allocation framework

The generic patterns are often called analysis patterns. For example one such pattern is Party (Person or Company), Place, Thing, Event. This forms the basis of event planning or resource scheduling. However these may be far too abstract and generic to be used in an application.

Advantages and Disadvantages of Building Frameworks

The advantages of building a framework include:

- A reduction in cost and development time for each specific domain version using the framework

  Many of the time consuming development aspects, for example, the generic requirements gathering, requirement analysis, design, and testing have been completed and are contained in the framework. Therefore the cost and time to develop the specific version may be significantly reduced.
- May result in a competitive advantage

  If your time to market and cost are reduced, you can achieve a significant advantage over your competitors' offerings.
- Developers can focus on the differences between the specific domain and the framework
- Frameworks usually include good OO practices and patterns

The disadvantages of building a framework include:

- Can be expensive to build

  The cost to build a generic framework will be high because the developer has to consider all the possible use cases in which the framework will be used and provide a flexible framework to accommodate the specific domain versions

- Requires an excellent knowledge of all the domains that the framework will be used for

  You need access to domain experts in all of the specific domains that the framework will be used for.
- The code may become bloated due to the one size fits all approach of a framework
- The framework code and specific version code combined will always be greater than a version written from scratch
- Frameworks may be difficult to learn
- You are restricted by the infrastructure and cannot usually modify the infrastructure files

  If the developers of the framework did not consider all the possible use cases, then the framework may not fit. For example, consider a vehicle domain framework envisaged to be used for vehicles of land, water, or air. If the framework used an attribute of weight this would cause a problem in a case where the framework gets used for Space vehicles, where an attribute of mass would be more appropriate.

Summary

In this module, you were introduced to the basic concepts of:

- Frameworks
- Using existing frameworks
- Creating domain neutral and domain specific frameworks

Module 16

Course Review

Objectives

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

- Review the key features of object orientation
- Review the key UML diagrams
- Review the Requirements Analysis (Analysis) and Design workflows

Overview

During this module, your instructor will facilitate an interactive review in which you will recall and review the key topics covered in the course.

This is your opportunity to review the key topics covered in the course.

This course covered three main aspects that ran concurrently throughout the course:

- Object orientation
- UML diagrams
- The development process, focusing on the Analysis and Design workflows

Reviewing Object Orientation

Discussion – In this instructor-facilitated discussion, you will recall and review the key object-oriented (OO) concepts and terminology covered in the course.

This page is intentionally left blank

OO Concepts and Terminology: A Recap

The following are the key OO concepts and terminology that were covered in this course:

- Object

  Represents a runtime instance of a class.
- Class

  Represents a template for describing objects that share common data, structure, and behavior.
- Abstraction

  Represents a real world object with irrelevant details either hidden or removed.
- Encapsulation

  Represents an object's data and methods that are not directly accessible from outside of the object.
- Inheritance

  Represents a class that can be defined in terms of extending an existing class.
- Abstract class

  A class from which objects cannot be instantiated, often because the class is not fully defined in terms of its implementation.
- Interface

  Represents a classification of a set of methods that have no method implementation.
- Polymorphism

  Represents a mechanism where the runtime behavior is determined by the form of the object. However, there are other less object-oriented definitions of polymorphism.
- Cohesion

  Represents a relative measure of how well a class, component, or method supports a single purpose.
- Coupling

  Represents a relative measure of the dependence of one component on other components.

- Class associations and object links

  Represent the relationships between classes and objects respectively.
- Delegation

  Represents the delegation of the behavior of a method to other methods that are in the same class or in another class.
- Design pattern

  Represents a known solution to a common design problem. This solution can be customized to meet the specific requirements of the problem.

Reviewing UML Diagrams

Discussion – In this instructor-facilitated discussion, you will recall and review the key UML diagrams and their purposes.

This page is intentionally left blank

UML Diagrams: A Recap

The following are the key UML diagrams that were covered in this course:

- Use Case diagram

  Represents the set of high-level behaviors that the system must perform for a given actor.
- Class diagram

  Represents a collection of software classes and their interrelationships.
- Object diagram

  Represents a runtime snapshot of software objects and their interrelationships.
- Communication diagram (formerly Collaboration diagram)

  Represents a collection of objects that work together to support some system behavior.
- Sequence diagram

  Represents a time-oriented perspective of communication between objects.
- Activity diagram

  Represents a flow of activities that might be performed by either a system or an actor.
- 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.
- Component diagram

  Represents a collection of physical software components and their interrelationships.
- Deployment diagram

  Represents a collection of components and shows how these are distributed across one or more hardware nodes.
- Package diagram

  Represents a collection of other modeling elements and diagrams.

The following UML diagrams were not formally covered in this course:

- 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.

- 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.

- 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).

- Profile diagram

  Might define additional diagram types or extend existing diagrams with additional notations.

Figure 16-1 shows a pictorial representation of the 14 UML 2.2 diagrams.

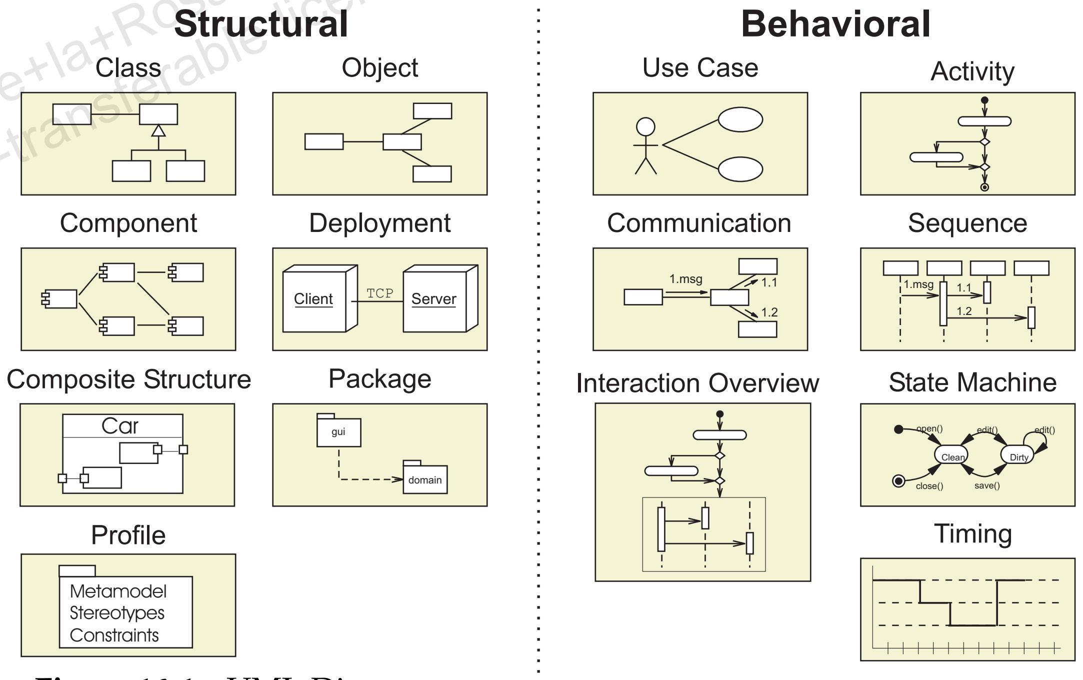


Figure 16-1 UML Diagrams

Reviewing the Development Process

Discussion – In this instructor-facilitated discussion, you will first recall and review the key workflows of the entire development process. Then, you will focus on reviewing the key activities and artifacts of the Analysis and Design workflows.

This page is intentionally left blank

The Development Process: A Recap

The following workflows were covered in the course:

- 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.

Although the course discussed all of the above workflows, it focused mainly on the Analysis and Design workflows.

The Analysis and Design Workflows: A Recap

The following diagrams are just one interpretation of how to implement a development process.

Figure 16-2 shows a high-level view of one iteration covering the Analysis, Design and Architecture workflows. The Architecture workflow has been shown as a running concurrently alongside the Analysis and Design workflows.

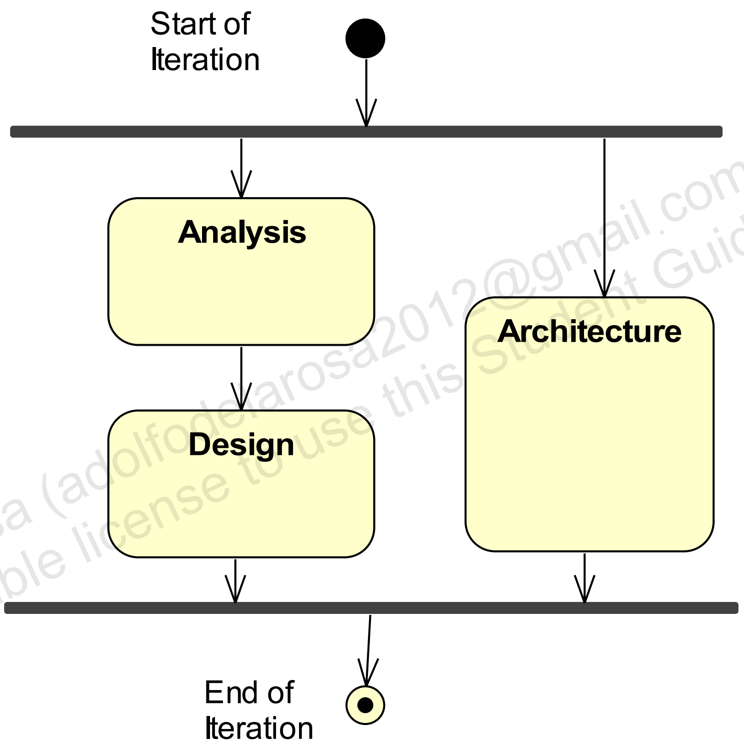


Figure 16-2 Example of a High-Level View of the Analysis, Design, and Architecture Workflows

Figure 16-3 shows an example of the activities and artifacts covered in one iteration of the Analysis workflow.

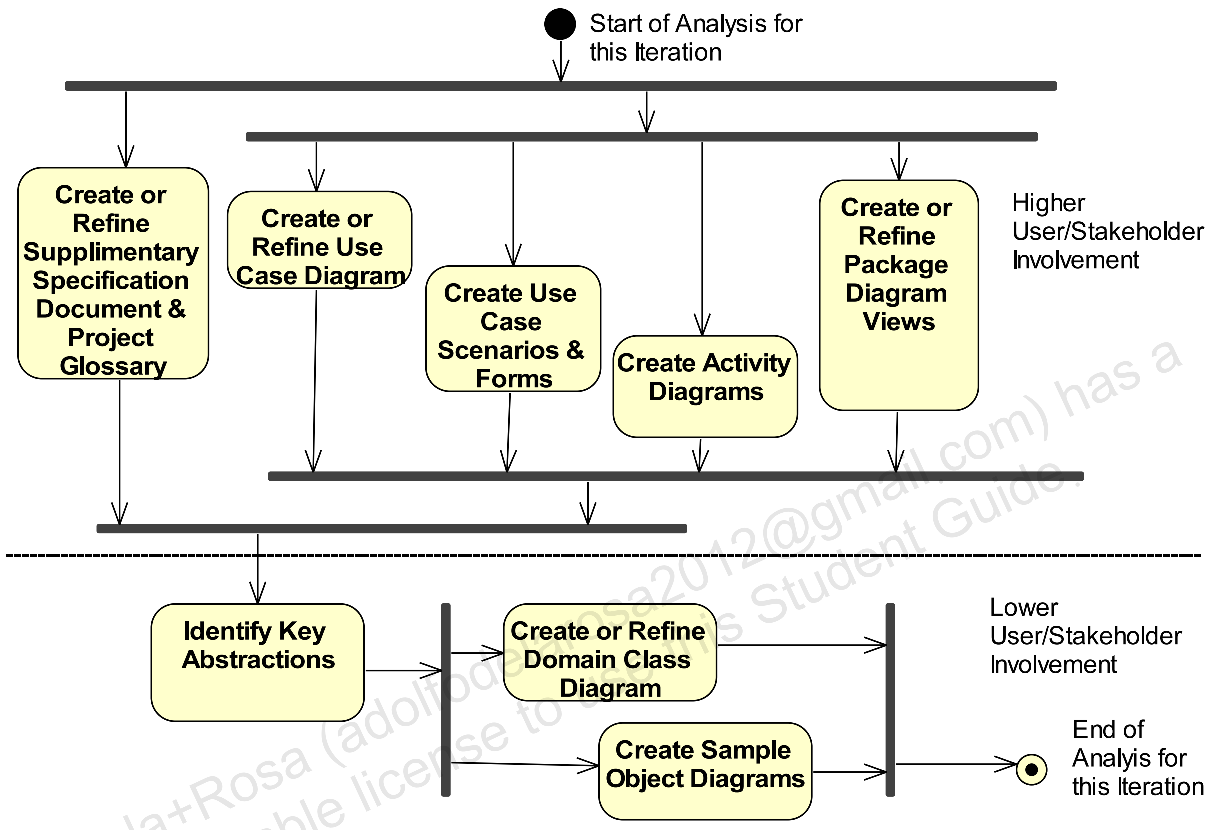

Start of Analysis for this Iteration
Create or Refine Supplimentary Specification Document & Project Glossary
Create or Refine Use Case Diagram
Create Use Case Scenarios & Forms
Create Activity Diagrams
Create or Refine Package Diagram Views
Higher User/Stakeholder Involvement
Identify Key Abstractions
Create or Refine Domain Class Diagram
Create Sample Object Diagrams
Lower User/Stakeholder Involvement
End of Analyis for this Iteration

Figure 16-3 Example of Activities in the Analysis Workflow

Figure 16-4 shows an example of the subactivities that occur during the Identify Key Abstraction activity shown in Figure 16-3 on page 16-15.

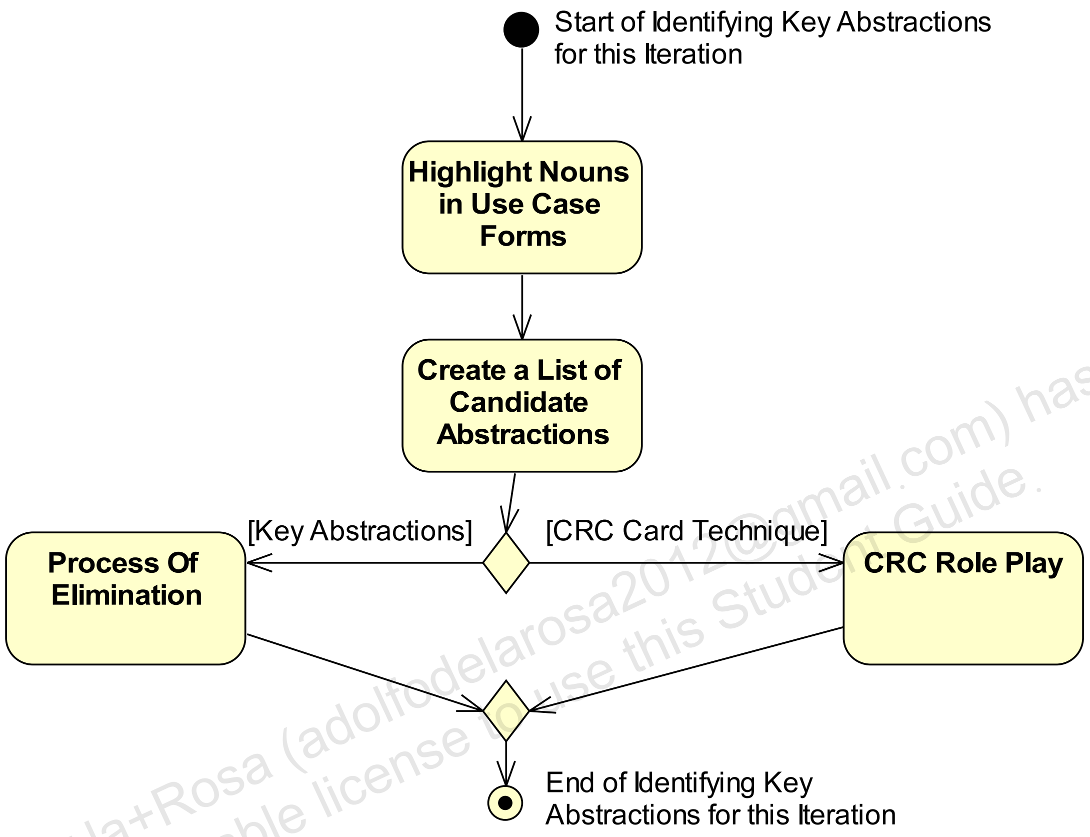

Start of Identifying Key Abstractions for this Iteration
Highlight Nouns in Use Case Forms
Create a List of Candidate Abstractions
[Key Abstractions]
[CRC Card Technique]
Process Of Elimination
CRC Role Play
End of Identifying Key Abstractions for this Iteration

Figure 16-4 Example of the Identify Key Abstractions Activity

Figure 16-5 shows an example of the activities and artifacts covered in one iteration of the Design workflow.

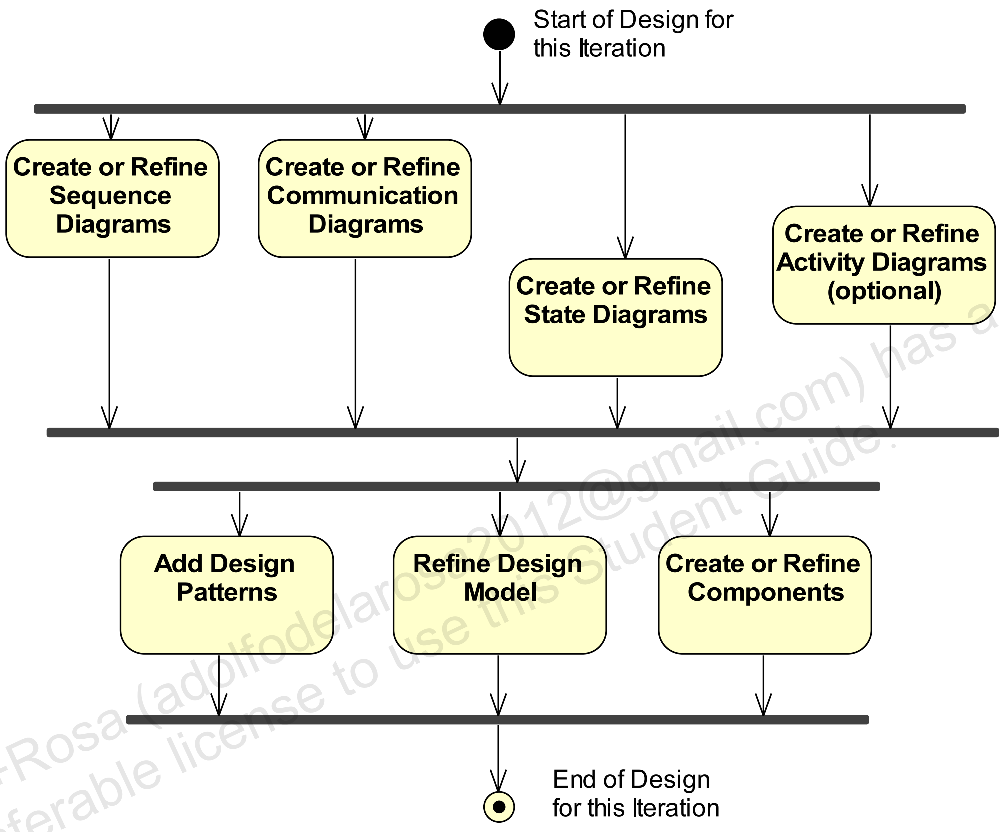

Start of Design for this Iteration
Create or Refine Sequence Diagrams
Create or Refine Communication Diagrams
Create or Refine State Diagrams
Create or Refine Activity Diagrams (optional)
Add Design Patterns
Refine Design Model
Create or Refine Components
End of Design for this Iteration

Figure 16-5 Example of Activities in the Design Workflow

Summary

In this module, you reviewed the three main aspects that ran concurrently throughout the course:

- Object orientation
- UML diagrams
- The development process, focusing on the Analysis and Design workflows

Drafting the Development Plan

Objectives

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

- Describe how the SunTone Architecture Methodology relates to constructing a development plan
- List the resources (namely developers) and the skills required to build the system
- Select use cases for each iteration based on a set of criteria
- Document the development plan using a UML package diagram

Relevance

Discussion – Some of the hardest decisions in constructing software are: Where does the team start? How often should a team *release* the system during development? The development plan solves these issues by providing a road map to the construction of the system.

- How do you (or your project manager) decide how many developers are required to build a system?

  ____

  ____

  ____

- How do you (and your development teams) decide what functionality to build first?

  ____

  ____

  ____

- Have you worked on a team that used iterative development? What benefits do you see to this strategy?

  ____

  ____

  ____

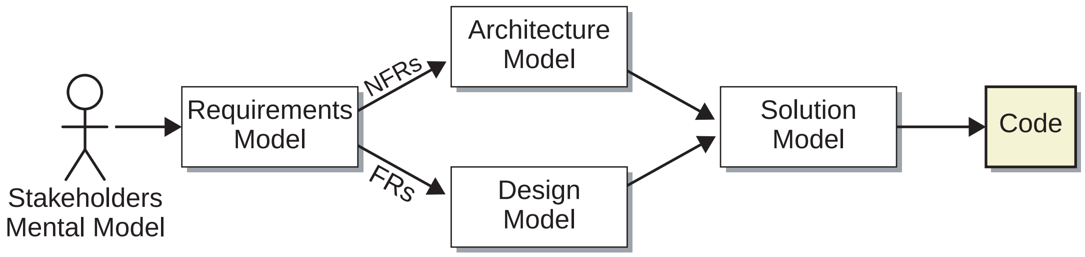
Additional Resources

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

- Booch, Grady. *Object Solutions (Managing the Object-Oriented Project).* Reading: Addison Wesley Longman, Inc., 1994.
- Brooks, Frederick. *The Mythical Man-month (anniversary edition)*. Reading: Addison Wesley Longman, Inc., 1995.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999.

Process Map

This appendix describes a step in the Implementation workflow: creating the development plan. Figure A-1 shows the activity and artifact discussed in this appendix.



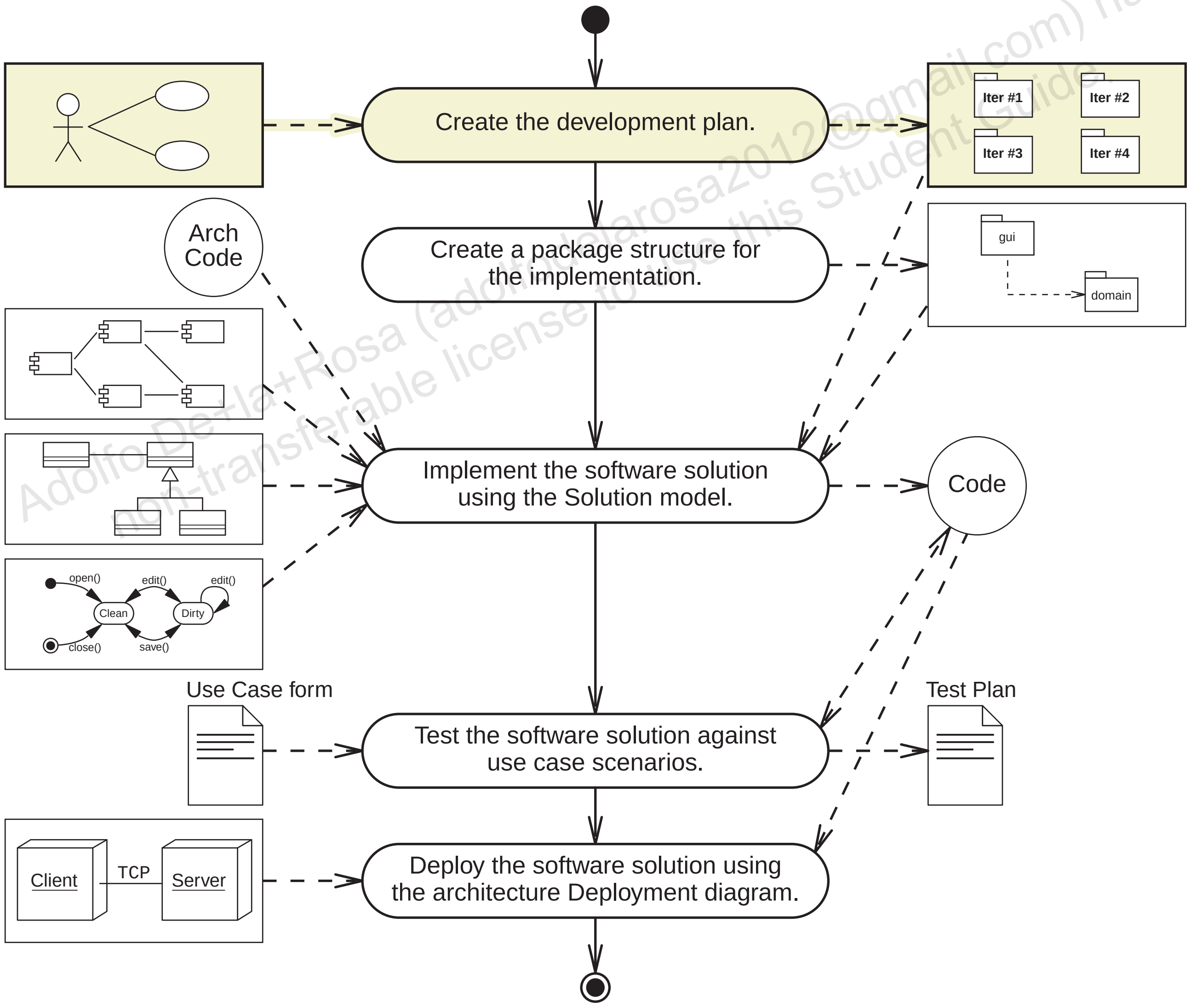

Create the development plan.
Iter #1
Iter #2
Iter #3
Iter #4
Arch Code
Create a package structure for the implementation.
gui
domain
Implement the software solution using the Solution model.
Code
open()
edit()
edit()
Clean
Dirty
close()
save()
Use Case form
Test Plan
Test the software solution against use case scenarios.
Client
TCP
Server
Deploy the software solution using the architecture Deployment diagram.

Figure A-1 Implementation Workflow Process Map

Describing How SunTone Architecture Methodology Relates to the Development Plan

SunTone Architecture Methodology is compatible with the Unified Process. Relative to creating a development plan, these are the most important features of the methodology:

- Four phases: Inception, Elaboration, Construction, and Transition

  The life cycle of the project is split into four large phases. During the Inception phase you define the vision of the system. During the Elaboration phase you define the bulk of the system requirements and develop the architecture baseline. During the Construction phase you build the Beta system. Finally, during the Transition phase you deploy the production system.

- Iterative and Incremental

  In SunTone Architecture Methodology, development of the system is done incrementally starting with a small working system. This small system, called the Architecture baseline, is built by the architect (or the architect's team) during the Elaboration phase.

  The Construction phase has multiple, small iterations. Each iteration implements a small set of use cases. Each iteration ends with a working, integrated system. By keeping the iterations short, the development team and client see steady progress.

The development plan will partition the work of the development team into distinct iterations within the phases. Therefore, the plan is partitioned into phases and iterations.

Creating a development plan is challenging because:

- Every developer has their own speed.

  Developers are not interchangeable parts. Every person has different abilities, experience, and development speed. Estimates should be made by the developers themselves. This provides better estimates to the project manager. This also provides buy-in from the development team.

- The size of the team affects individual performance; larger teams add communication overhead.

  Small, focused teams tend to be most effective. Small might mean five plus or minus two developers. When a team becomes too large, the inter-team communication becomes a significant time issue.

However, some projects might have large, separable subsystems that can be developed in parallel by separate teams; each of these teams should be as small as possible.

- Project complexity adds to hidden infrastructure that is not obvious from the use cases.

  Being use-case-driven is only half the story when it comes to building a complex application because use cases only identify the functional requirements. A complex system might have significant non-functional requirements. The NFRs must be considered during planning as well as the FRs (embodied in the use cases).

  The purpose of the Architecture baseline is to identify and mitigate the complexity and risks of this hidden infrastructure.

- Estimates and project schedule should be reevaluated periodically.

  In an incremental project, the project schedule must be viewed as a changeable, *living* document. The project manager should review and update the schedule periodically, and at major milestones, such as at the end of the Inception and Elaboration phases.

Listing Needed Resources and Skills

Based on the FRs and NFRs from the SRS, the architect determines how many people are needed on the team and the skills required to develop the system.

- Development team size is determined by the size and complexity of the project.

  The greatest factors in determining the size of the development team are the size and complexity of the proposed system. However, it is important to realize that it is not so easy to increase the size of the team to reduce the length of the project life cycle. (see Brooks chapter 2)

- Skill set is determined by the architectural needs of the project.

  The architecture of the system will determine the skills that the development team must possess. For example, if the architecture requires database access, then at least one team member must have SQL experience.

These issues are discussed in more depth in the following sections.

Determining the Developers

Factors that determine the number of developers:

- Size and complexity of the use cases

  As mentioned previously, the size and complexity of a use case can be a factor in determining the team size because a complex use case tends to require more developers to build that part of the system in the short time frames devoted to an iteration.

  However, a large use case might be split across multiple iterations if there are separable parts (or flows) to the use case. For example, use case E6 "Generate Reports," might include a dozen or so distinct reports. The creation of the set of reports could be split across multiple iterations.

- Size and complexity of the architectural baseline

  If the required infrastructure is large or complex, then more developers might be needed to build that part of the system in the short time frames devoted to an iteration.

- Issues with legacy system integration

  A project that requires integration with a legacy system must have one or more developers devoted to this task. These developers should have experience with the legacy system.

- Duration of the project

  If the project is calendar-driven, then the team size might be increased to reduce the overall project schedule. However, you cannot increase the project team without complicating other issues such as inter-team communication, which will tend to extend the duration of the project.

- Modularity of the components

  Several developers can work on different components if the architecture partitions the system into many small modular components. If the architecture is not very modular, then these components must be worked on by a single team.

  Also, separation of components within specific architecture tiers facilitates clear job roles: business component developer, web component developer, web content creator, GUI developer, data access developer, and so on.

For the Hotel Reservation System, the following factors influenced the size of the development team:

- The size and complexity of the use cases is relatively small.

  Only a few use cases were large or complex, such as E1 "Managing Reservations" and E5 "Managing Reservations Online." These two use cases can also be split up into smaller activities, which makes partitioning the work easier.

- The project is not schedule-bound, so the duration of the project can remain flexible.

  The Hotel Reservation System project does not have a fixed deadline, so the duration of the project can be flexible. This means that a smaller team can be used to build the system.

- The system can be separated effectively into three general technical areas:
  - GUI development
  - Web application development
  - Business logic (plus infrastructure) development

Based on this information, the development team for this project will include one GUI developer, one Web application developer, and one business component developer.

Determining the Skill Set

The skill set of the team is important to the success of the project. Factors that determine the skill set are based on high-level architectural decisions:

- Programming language and platform

  Will the project be developed in an OO language? Has the SRS constrained the specific language for the project? What operating system (OS) or hardware platform will be used? The answers to these questions will determine elements of the team's skill set.
- Specific technologies:

  Likewise, does the SRS (and the architecture baseline) constrain the technologies with which the team must be skilled? Based on the different architectural tiers, here is a sample of technologies that might be needed for a project.
  - Client tier: HTML, JavaScript, Swing, VB, and so on
  - Presentation tier: servlets, JSP, PHP, Perl, ASP, and so on
  - Business tier: RMI, EJB, CORBA, and so on
  - Integration tier: JDBC, XA, MOM, and so on
  - Resource tier: DBMS, EIS, and so on
- Development environment (tools, IDE, and so on)

  What tools will the team use? This is often in control of the development team (and its management), so this is not as critical.

Here is an example set of skills for the Hotel Reservation System project:

- J2SE (v1.3) is the selected platform

  Java technology will be used throughout the project. The Java platform was selected to provide a high degree of portability as well as great n-tier technologies such as web components and RMI.
- Client tier:

  There will be three applications: WebPresenceApp, KioskApp, and HotelApp. The first two are web applications and the second is an intranet GUI application. These are the required technologies:

  - HTML (for WebPresenceApp and KioskApp)
  - Swing (for HotelApp GUI)
- Presentation tier: servlets and JSP pages

  The WebPresenceApp requires Presentation tier (web) components. This will be Java technology servlets and JavaServer Pages.
- Business tier: RMI

  RMI technology is used to communicate from the Client and Presentation tiers to the Business tier. The components on the Business tier will be standard Java objects.
- Integration tier: JDBC/SQL

  The Integration tier will require JDBC technology and SQL to communicate to the database.
- Resource tier: PostgreSQL DBMS

  The Resource tier will use a relational database. The Open Source PostgreSQL system will be used.

Selecting Use Cases for Iterations

Each iteration should deliver a clearly defined portion of the behavior of the complete system. Therefore, complete use cases should be assigned to specific iterations. The following are factors that help determine how to partition use cases into iterations:

- Determine the criteria for grouping use cases into iterations.

  Each project will have different criteria for prioritizing use cases. This is described in the next section.

- Estimate the development duration for each use case.

  The development team must estimate the construction time for each use case. These estimates must be done by the developers themselves, because every developer has her own speed.

- Estimate the recommended length of each iteration.

  Estimating the length of an iteration is largely based on how many iterations the client wants. Some clients might want to see numerous iterations if they are concerned about the project staying on track. Other clients might be totally comfortable with the development team and require fewer iterations.

  Typically, the length of an iteration is between a few weeks to a few months.

- Determine which use cases will be handled in which iteration within which phase of the project.

  Use cases are assigned to iterations based upon the available resources and the prioritization of the use cases.

Partitioning use cases is described in more detail in the following sections.

Criteria for Prioritizing Use Cases

Each project will have different criteria for prioritizing use cases. The following are a few factors to help you determine the priority of each use case:

- Risky use cases should be handled as early in the project as possible.

  The SunTone Architecture Methodology service recommends all risky use cases be developed before the end of the Elaboration phase.

- Architecturally significant use cases should be handled as early as possible.

  An architecturally significant use case is one that presents a technical risk to the project. The most common technical issues include: concurrency requirements, timing or performance requirements, and legacy integration requirements.
- Essential use cases should be handled as early as possible.

  The priority of the use case from the SRS also helps to determine how to group the use cases into iterations. Essential use cases should be handled very early in the development. These should take precedence over High-value and Follow-on use cases, except if a High-value use case is also risky.
- High-value and Follow-on use cases can be distributed in the Construction phase, with Follow-on use cases being handled last.

  Leave the use cases that are deemed unimportant to this version of the system to the end of the project.

There might be other reasons to give a use case a higher priority; projects can be as flexible as possible. This is an important side-effect of using iterative development.

Estimating Development Time for Use Cases

Each development team must be able to estimate the development time for a given use case. The following are a few factors to help you understand how to approach estimations:

- Efficient tier separation (a developer for each major tier function)

  It helps to have clear tier boundaries (client tier, application tier, and integration tier) for a project. Typically, a team will have a single developer for a given tier. This configuration is useful because it enables the team to focus on a specific area of expertise, such as GUI development or database integration.

Note – An alternative strategy is to assign a use case to a single developer. This developer would implement the use case throughout every tier: client/presentation, business, integration, and resource. This strategy works well when every developer on the team has all of the required skills to work in all tiers.

- Skill and *speed* of the individual developers

  Every developer has a certain level of experience; the more experience the developer is the faster the developer can code. However, some developers are inherently faster programmers than others. Speed and skill are the two main factors for an individual developer to estimate how fast that developer can develop a given use case.

- Complexity and amount of infrastructure development

  Not all programming is confined to implementing the functional requirements of a use case. Some programming involves building the infrastructure within which the functional components must fit. The amount of infrastructure development will likely be higher in the beginning of the project than later in the project.

- Number of scenarios in a use case

  The number and complexity of the scenarios for a use case can help identify the difficulty of the implementation of the functional components for the given use case. For example, the scenarios might identify how many different paths there are through the user interface. This might identify how many screens are involved or how many major functions in the business services need to be coded.

For the Hotel Reservation System, use case “E1: Manage Reservation:”

- Is partitioned into four tiers: Client, Business, Integration, and Resource

  The client tier will be a Swing GUI. The application tier requires service components to coordinate the use case functions as well as a variety of entity components for the domain objects. The integration tier requires the development of data access components to persist the entities within the database. The resource tier requires the specification of the database tables for the entities.

- Requires significant infrastructure development

  Because this will be the first use case to be developed, there will be a significant amount of infrastructure development. On the client tier, the fundamental security mechanism must be implemented to support login. On the application tier, the RMI server must be implemented. On the resource tier, the PostgreSQL DBMS must be configured and populated with the initial schema.

- Includes three major scenarios (create, update, and delete)

  On the client tier, it is likely that a single GUI window could be used to implement all three scenarios. On the business tier, the service component must have all of the CRUD functionality to manage reservation entities as well as to create and retrieve customers.

For the Hotel Reservation System, use case "E1: Manage Reservation:"

- GUI developer: 6 weeks

  The GUI developer will spend the first week prototyping the fundamental GUI layout that will be used for all HotelApp screens. The second week will be spent developing the main screen and the login dialog box. The remaining weeks will be spent developing the reservation form and any other related dialog boxes for this use case.

  The breakdown of tasks include:

  - GUI layout design: 1 week
  - MainUI screen: 1 week
  - ResvUI, CustMgtUI, and PayMgtUI: 4 weeks
- Web developer: N/A

  This use case does not contain a web interface, so there is no work for this developer.
- Business (and integration) developer: 4 weeks

  This developer, Annie Codewalker, is a programming wizard, and she has estimated four weeks for the following: RMI service component for the use case, entity classes, DAO classes, database schema, and RMI server.

  The breakdown of tasks include:

  - Three remote services: 2 weeks
  - Five entities: 0.5 weeks
  - Five (partial) DAOs: 2.5 weeks

These durations are determined by the two developers by estimating the number of components required by the given use case. This information can be determined from the Solution model.

Determining the Duration of Each Iteration

Each project will have different criteria for determining the duration of an iteration. The following are a few factors to help you determine the duration:

- Long enough to develop the longest use case

  Any given iteration must be long enough to include development time for the longest use case in that iteration.
- Short enough to see results and maintain the project's momentum

  Iterations should be kept as short as possible to show results to the client and to build momentum within the team.
- Iteration Duration = For each UC, sum of UC duration

  As a rough estimate, the duration of an iteration is the duration of each use case specified for that iteration.
- UC Duration = For each developer, maximum development time for the use case

  The duration for a use case is the maximum development time of all developers working on that use case.

Table A-1 provides an example estimate for the first iteration of the Hotel Reservation System.

Table A-1 Developer Estimates for Iteration 1 Use Cases

Use Case	**GUI Developer**	**Web Developer**	**Business Developer**
E1: Manage Reservation	6 weeks	N/A	4 weeks
E5: Manage Reservation Online	N/A	4 weeks	1 week
Developer Workload	6 weeks	4 weeks	5 weeks

Each developer works in parallel, but a given developer cannot work on two use cases in parallel. So for example, the GUI and web developers can work in parallel, one on E1 and the other on E5. The Business component developer needs to spend four weeks on E1 and then one week on E5 for a total time of five weeks. Therefore, iteration 1 will take six weeks total.

Grouping Use Cases Into Iterations

These are a few factors to help you determine how to group use cases into iterations and phases:

- Group risky and architecturally significant use cases into iterations in the Elaboration phase.

  During the Elaboration phase, you construct the Architecture baseline and eliminate the risks in the projects. The architecturally significant and risky use cases must be scheduled in the iterations of the Elaboration phase.

- Group all (remaining) Essential and High-value use cases into iterations in the Construction phase.

  As mentioned earlier, schedule all Essential and High-value use cases as early in the Construction phases as possible.

- Leave the Follow-on use cases for the end of the Construction phase.

Figure A-2 shows an example schedule of iterations in the Elaboration and Construction phases for the Hotel Reservation System.

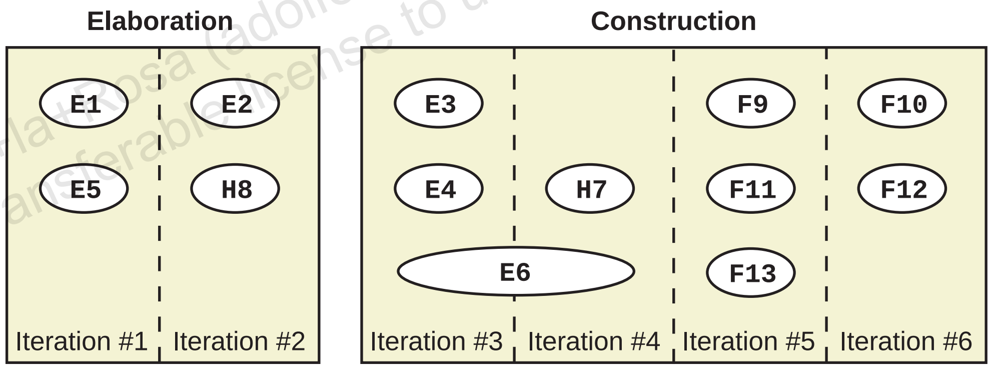


Figure A-2 Iterations for the Hotel Reservation System

Documenting the Development Plan

There are many ways to document the development plan. Most project managers use a tool that support GANTT charts. However, these are often detail-oriented. A UML package diagram can be an alternative:

- A UML package diagram shows the iteration and phase groupings.

  You should provide the customer with a high-level view of the iteration schedule. This can be done pictorially by providing a UML package diagram of the use cases for each iteration.

- The UML package notation.

  A UML package icon can represent a group of other modeling elements. Figure A-3 shows two variations on the package icon.

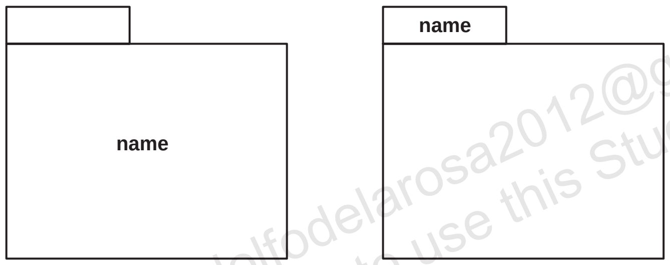


Figure A-3 Two Variations on the UML Package Icon

- The package name can be placed in the body box or in the name box.

  Use the first view (with the name in the body box) when you do not want to show the contents of the package, but just want to name the package. Use the second view (with the name in the name box) when you do want to show the contents.

- You can place any UML entity in a package, including other packages.

  The contents of a package icon are completely unrestricted. You can include single UML entities, partial diagrams, complete diagrams, or even nested packages.

Figure A-4 demonstrates the use of a UML package diagram to provide a high-level view of the schedule for the Elaboration phase for the Hotel Reservation System.

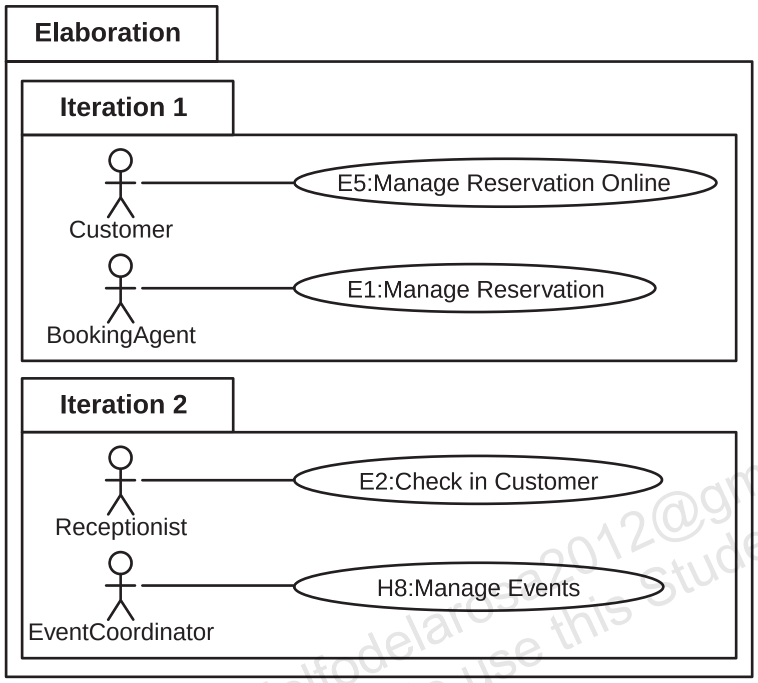


Figure A-4 Package Diagram of the Elaboration Phase for the Hotel Reservation System

Summary

In this appendix, you were introduced to a technique for estimating a use case development effort as well as scheduling iterations. Here are a few important concepts:

- The members of the development team are constrained by project size and the skill set required for the technologies used by the project.
- Development is organized into iterations. Each of which acts as a mini-project.
- Use cases are partitioned into iterations. The duration of the iteration is determined by the time estimates for each developer for each use case.
- The development plan can be represented using a UML package diagram showing iterations as packages containing the use cases to be developed in that iteration.

Constructing the Software Solution

Objectives

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

- Define a Java technology package hierarchy for the Solution model
- Recognize Java technology code that satisfies the elaborated Domain model

Process Map

This appendix describes a step in the Implementation workflow: implementing the solution code. Figure B-1 shows the activities and artifacts discussed in this appendix.

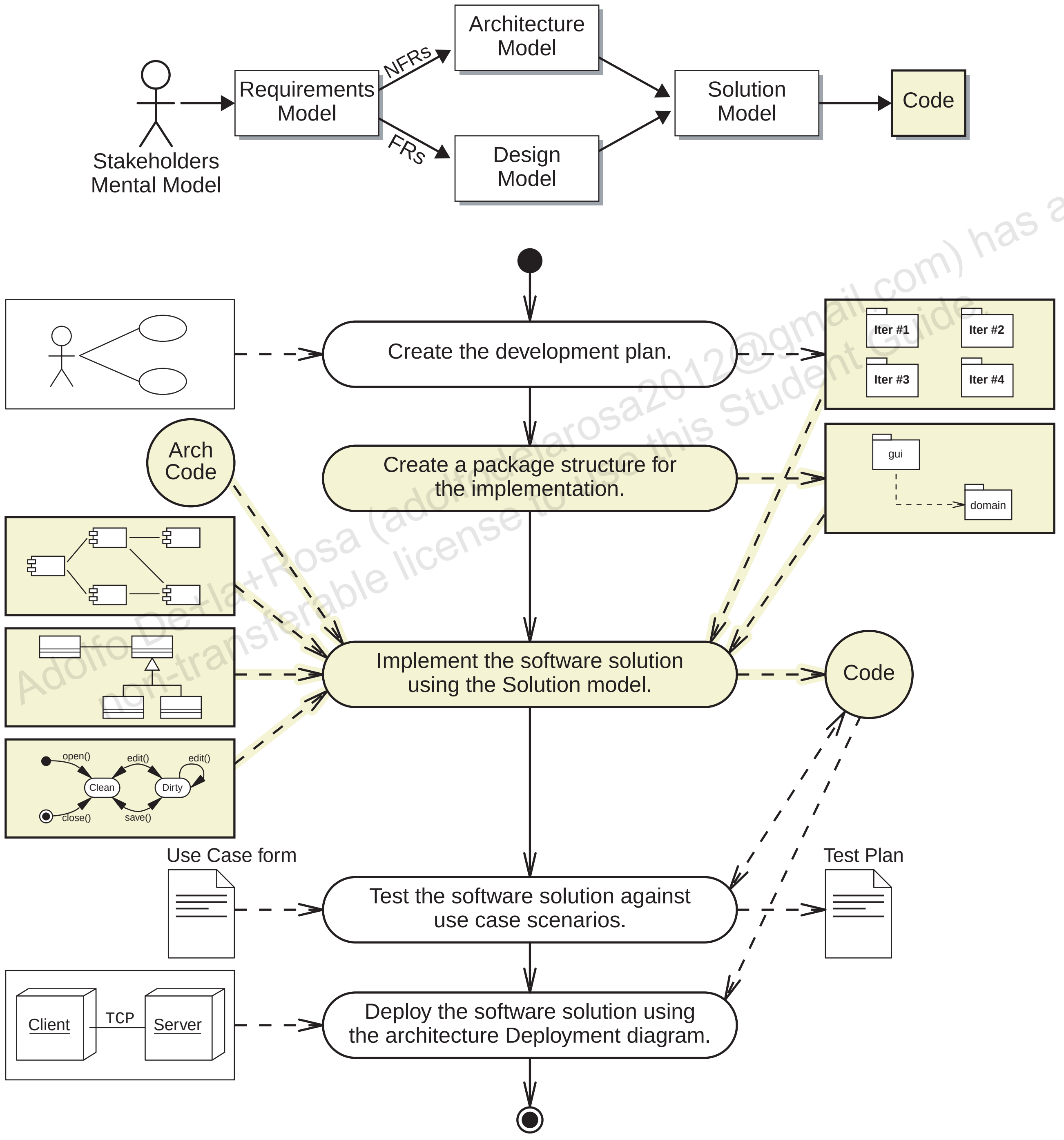

Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code
Iter #1
Iter #2
Iter #3
Iter #4
Arch Code
gui
domain
open()
edit()
edit()
Clean
Dirty
close()
save()
Use Case form
Test Plan
Code
Client
TCP
Server
Create the development plan.
Create a package structure for the implementation.
Implement the software solution using the Solution model.
Test the software solution against use case scenarios.
Deploy the software solution using the architecture Deployment diagram.

Figure B-1 Implementation Workflow Process Map

Defining a Package Structure for the Solution

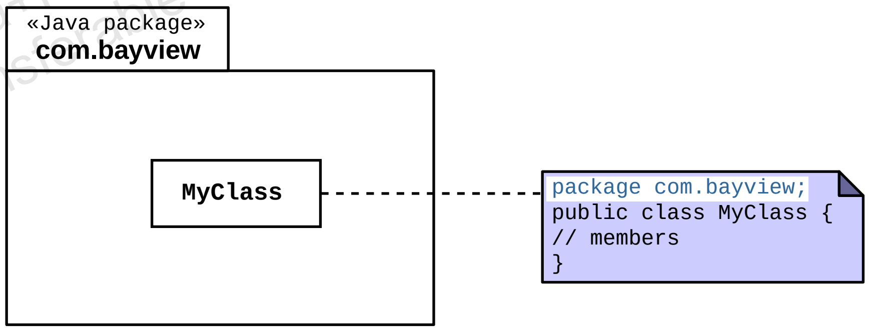
This section describes the concepts and principles behind defining a useful package structure for a software solution. This section is grouped into the following subsections:

- Representing Java technology packages in UML
- Applying the principles of package dependencies
- Isolating subsystems and frameworks into packages
- Developing a package structure that satisfies the Solution model

Using UML Packages

A UML package is a modeling construct for grouping other UML diagrams or elements.

A UML package can also represent a Java technology package (a group of related classes). Figure B-2 illustrates an example of a UML package that models a Java technology package. The code in the annotation demonstrates the use of the `package` statement to declare that the `MyClass` class is part of the `com.bayview` package.



```
package com.bayview;
public class MyClass {
// members
}
```

Figure B-2 Example Code for a Java Technology Package

Applying Package Principles

This subsection introduces the following package principles:

- Package dependencies
- Common Closure Principle (CCP)
- Acyclic Dependency Principle (ADP)

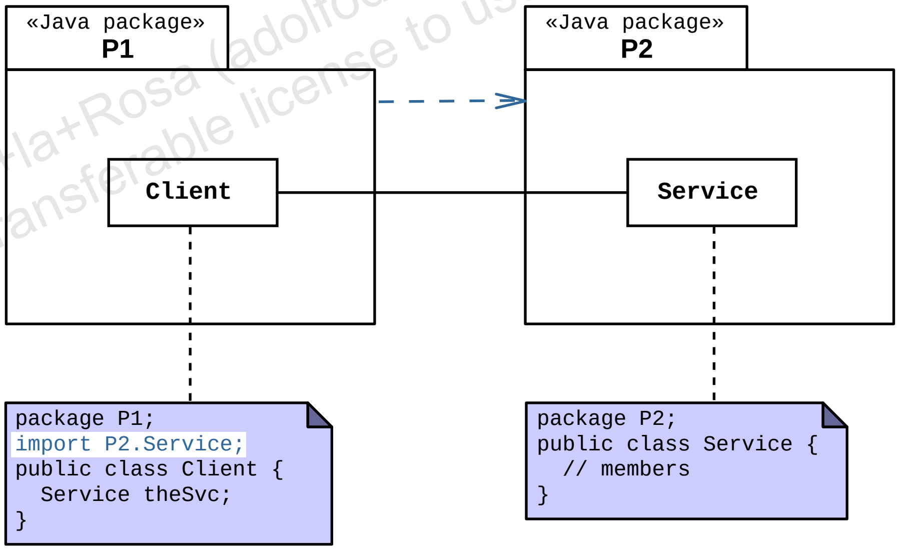
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

All of these principles are just guidelines for good design and that they require trade-offs. These principles should not be considered unbreakable rules. If your project has a requirement that leads you to break one or more of these guidelines, then do so.

Simple Package Dependencies

"If changing the contents of a package P2 (might) impact the contents of another package P1, then we can say that P1 has a package dependency on P2." (Knoernschild page 24)

All package principles are defined in terms of package dependencies. Figure B-3 illustrates a simple package dependency. Package `P1` depends on package `P2` because the `Client` class (in `P1`) uses the `Service` class (in `P2`). The code in the annotations demonstrate the use of the `import` statement to allow a class definition in one package to gain access to a class in another package.



```
package P1;
import P2.Service;
public class Client {
  Service theSvc;
}
```

```
package P2;
public class Service {
  // members
}
```

Figure B-3 An Example Package Dependency

Common Closure Principle

"Classes that change together, belong together." (Knoernschild page 26)

CCP recognizes package cohesion, emphasizing the overall services offered by the entire package.

CCP tends to group packages by Separation of Concerns. For example:

- Grouping web components for a single use case together

  Web components make up the Boundary elements for use cases that require a Web application. For example, permitting Bay View customers to make reservations online. If there are changes to the "Manage a Reservation Online" use case, then it is likely that several web components for that use case will need to change. Therefore, CCP suggests that all of the web components for this use case be grouped together.

- Grouping related Domain classes together

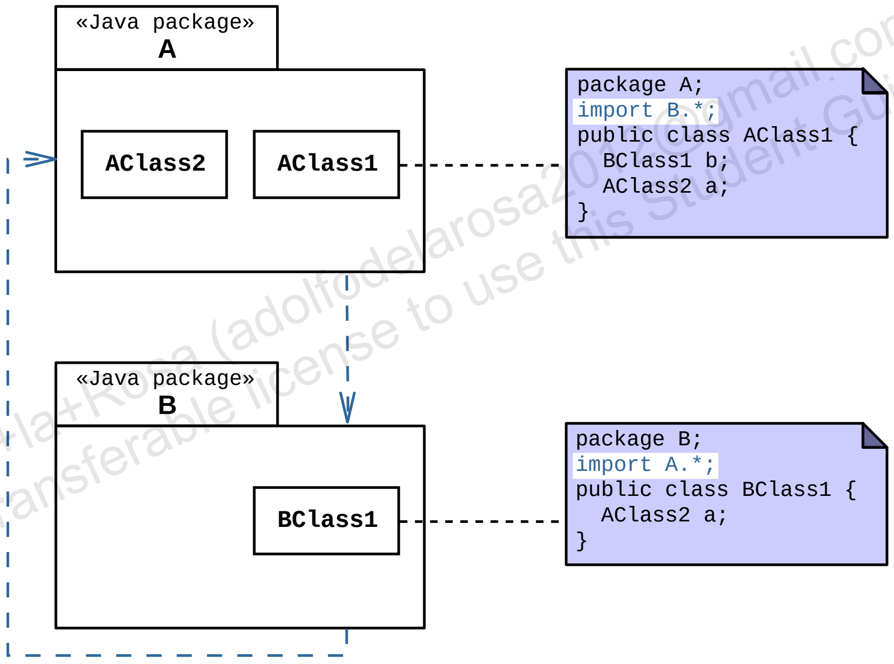
  For small systems, all of the Domain entities could be grouped into a single package. Large systems might have groups of entities that are highly coupled. Using CCP, these groups would be separated into their own packages.

  For example, in the Hotel Reservation System, the `Reservation`, `Customer`, `Room`, and `Payment` classes might all be grouped together. Whereas the classes that support the catering side of the business might be grouped into a different package.

Acyclic Dependency Principle

> "The dependencies between packages must form no cycles." (Knoernschild page 27)

Suppose that a class, `AClass1` in package `A`, uses two other classes: `AClass2` in package `A` and `BClass1` in package `B`. And further suppose that `BClass1` uses `AClass2`. Figure B-4 illustrates this situation. `AClass1` is tightly coupled with `BClass1` and `BClass1` is tightly coupled with `AClass2`. This implies that package `A` is tightly coupled with package `B` and package `B` is tightly coupled with package `A`. This coupling forms a cycle in the package dependency diagram in Figure B-4 and is a violation of the ADP.



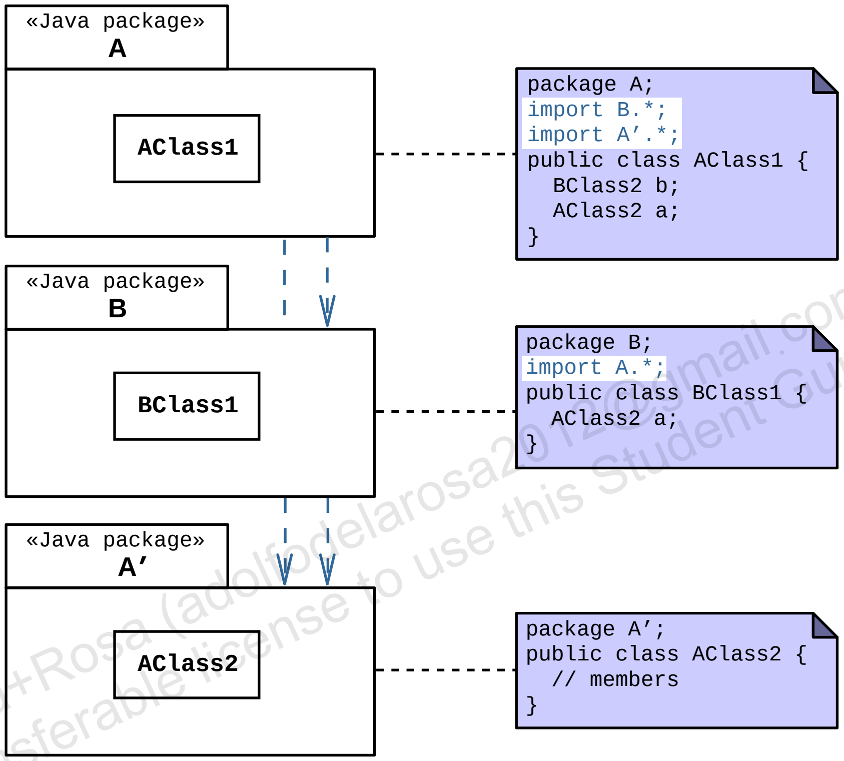
```
package A;
import B.*;
public class AClass1 {
  BClass1 b;
  AClass2 a;
}
```

```
package B;
import A.*;
public class BClass1 {
  AClass2 a;
}
```

Figure B-4 An Example Violation of the ADP

Violations to ADP hinder the effectiveness of the Common Closure Principle because changes in one package might affect changes in all of the packages in the cycle with the original package. The goal is to attempt to isolate the effects of changes between packages. Therefore, you should eliminate package dependency cycles. One way to do that would be to lump all of the packages in a cycle into a single package, but this approach has large consequences.

Alternatively, you could refactor the set of classes in one (or more) packages in the cycle to eliminate the cycle. For example, you could move `AClass2` into a different package, say `A′` (pronounced A-prime). By doing this the cycle between package `A` and package `B` is broken. Now package `A` depends on packages `B` and `A′`; and package `B` depends only on package `A′` (not package `A`). Figure B-5 illustrates this refactoring.



```
package A;
import B.*;
import A'.*;
public class AClass1 {
  BClass2 b;
  AClass2 a;
}
```

```
package B;
import A.*;
public class BClass2 {
  AClass2 a;
}
```

```
package A';
public class AClass2 {
  // members
}
```

Figure B-5 An Example Refactoring to Achieve ACP

Stable Dependencies Principle

"Depend in the direction of stability." (Knoernschild page 29)

SDP is an extremely useful and straightforward principle. It states that a given package should depend on more stable packages. For example, the most stable parts of a system tend to be the Domain entities because these do not change very often. However, business services and especially user interfaces that implement use cases tend to be less stable because business processes and UI designs (especially for the Web) tend to change frequently. Figure B-6 shows an example of SDP.

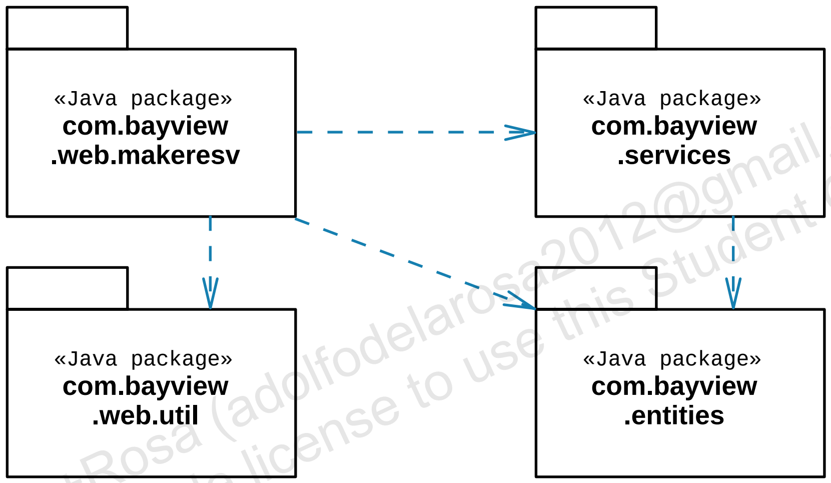


Figure B-6 An Example of the SDP for the Hotel Reservation System

Stability means a component is fixed, permanent, and unvarying.

- Business services and entities tend to be stable.
- Entities tend to be more stable than services.
- UI components tend to change.
- Utilities tend to be stable.

SDP (and CCP) compliment the Tiers partitioning of SunTone Architecture Methodology:

- Components closer to the resource tier tend to be more stable.
- Components closer to the client tier tend to be less stable.

Stable Abstractions Principle

> “Stable packages should be abstract packages.” (Knoernschild page 31).

This principle is related to the Dependency Inversion Principle for classes and components. The best example of the SAP is that you can develop a package that holds all of the interfaces for the business services of your system. The UI classes depend on interfaces of the services (especially in a distributed system); therefore, you should create a package that holds the set of service interfaces. It is also a common practice to package the concrete service classes (that implement the service interfaces) into a separate package. Figure B-7 illustrates an example of this package dependency structure.

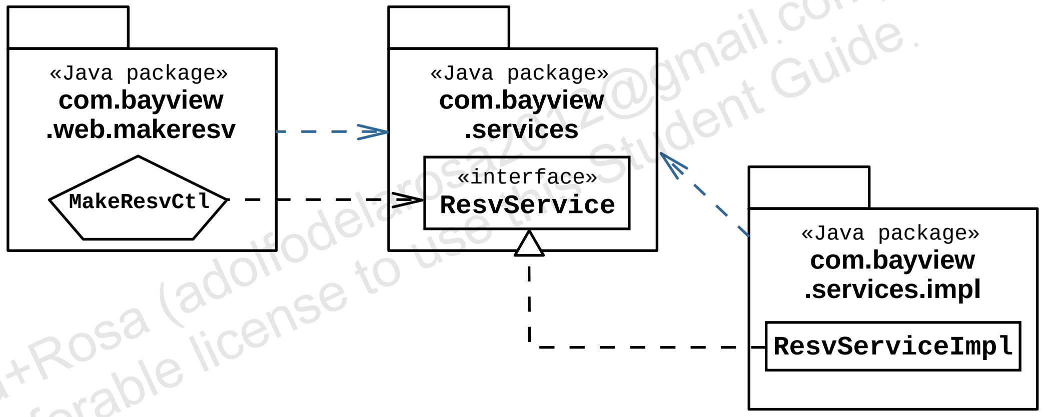


Figure B-7 An Example of the SAP for the Hotel Reservation System

The following are a few facts about the Stable Abstractions Principle:

- Less stable packages should depend on abstractions within another package, rather than concretions.
- Abstract classes and interfaces tend to change less frequently than their implementation classes.
- The formula for determining abstractness of a package is:

$$A = \frac{Na}{Nc}$$

Where, *Na* is the number of abstract classes and interfaces in the package and *Nc* is the total number of all classes and interfaces.

Isolating Subsystems and Frameworks

A subsystem is “A collection of modules, some of which are visible to other subsystems and other of which are hidden.” (Booch OOAD page 519)

A framework is “A collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms that clients can use or adapt.” (Booch OOAD page 514)

Subsystems and frameworks are usually placed in their own package or a set of packages. The I/O subsystem in the J2SE platform is grouped in the `java.io` (and now `java.nio`) package. The Abstract Window Toolkit (AWT) and Swing frameworks are in the `java.awt` and `javax.swing` packages, respectively.

Only public classes or interfaces are visible outside of the package. Therefore, subsystems and frameworks can expose only the classes that are required by the client of these subsystems by making those classes public. All other classes should be kept package-private.

Developing a Package Structure for the Solution Model

From the package dependency principles, the following are a few guidelines for creating a package structure:

- Group components of each tier into their own package.

  For example, the business services and entities should be in their own package. Likewise, the UI and DAO components should be in their own package.

  This recommendation is supported by the CCP and the SDP.
- Within a tier, group components that change together into their own package.

  For example, the client-tier components for a web application might be grouped together.

  Another example is to group all of the DAO classes for a given data storage mechanism together. So if you need an XML and DBMS data storage mechanisms, then the DAO classes for each of these would be in their own package.

- Group interfaces of a set of services into a package and their implementation into another package.

  This structure is important for distributed systems because the Client and Presentation tier components are designed to work with the service interfaces rather than with the service implementation classes.

- Isolate subsystems into their own packages.

  As mentioned in the previous section, subsystems should have their own package because the classes in a subsystem tend to change together. This guideline is an application of the CCP.

The following figures illustrate these guidelines relative to the Hotel Reservation System. Figure B-8 shows the packages of the Application layer of the system with their dependencies. Notice that the dependencies are acyclic and tend toward stability.

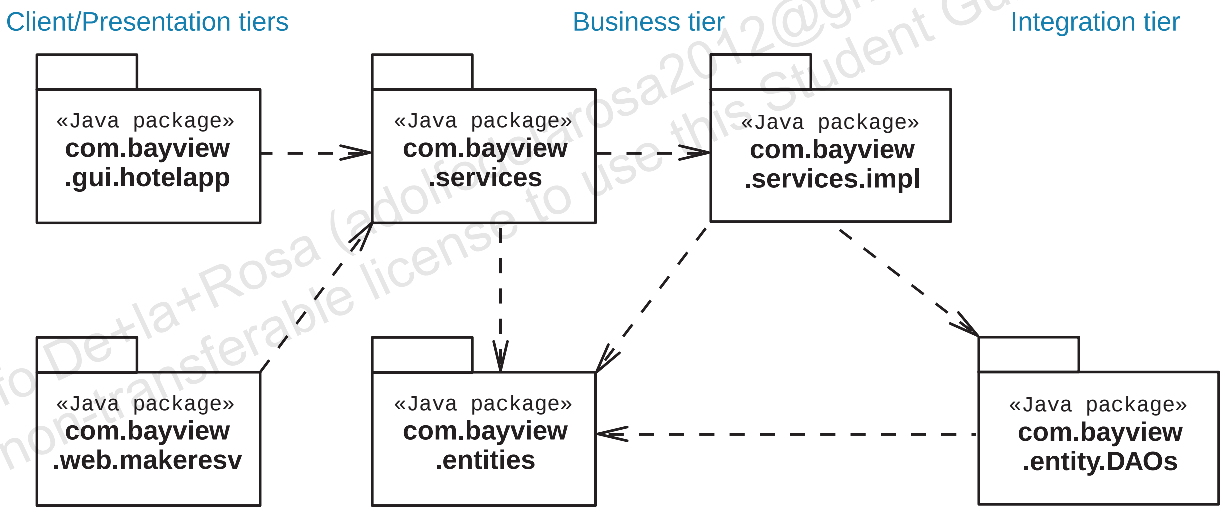

Client/Presentation tiers
Business tier
Integration tier
«Java package» com.bayview .gui.hotelapp
«Java package» com.bayview .services
«Java package» com.bayview .services.impl
«Java package» com.bayview .web.makeresv
«Java package» com.bayview .entities
«Java package» com.bayview .entity.DAOs

Figure B-8 HRS Package Dependencies Grouped by Tiers

Figure B-9 shows the dependencies between the packages of the Application layer and the packages of the Virtual Platform. For example, the web components in `com.bayview.web.createresv` depend on the servlet API package (`javax.servlet.http`).

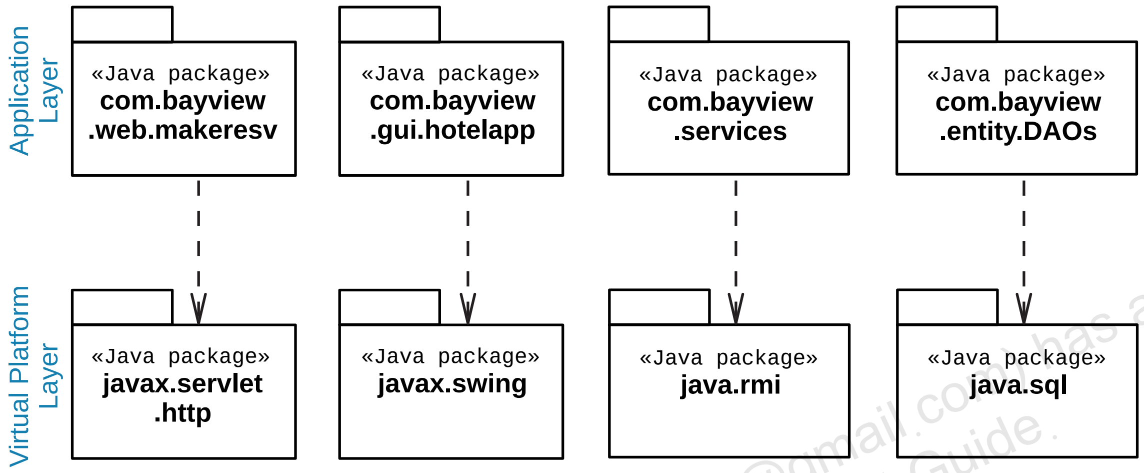

Application Layer
Virtual Platform Layer
«Java package» com.bayview .web.makeresv
«Java package» com.bayview .gui.hotelapp
«Java package» com.bayview .services
«Java package» com.bayview .entity.DAOs
«Java package» javax.servlet .http
«Java package» javax.swing
«Java package» java.rmi
«Java package» java.sql

Figure B-9 HRS Package Dependencies Between the Application and Virtual Platform Layers

Mapping the Domain Model to Java Technology Class Code

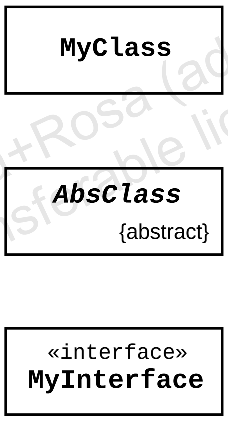
This section describes how to recognize Java technology code that implements the following modeling constructs in a UML Class diagram:

- Type information
- Attributes
- Associations

The following subsections illustrate these modeling constructs.

Type Information

There are three fundamental types of classes: a concrete class, an abstract class, and an interface. Figure B-10 illustrates Java technology code for each of these types.



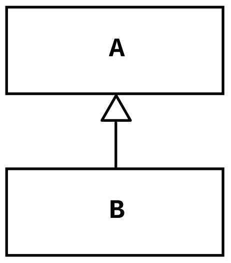
```
public class MyClass {
  // members
}

public abstract class AbsClass {
  // members
}

public interface MyInterface {
  // members
}
```

Figure B-10 Java Technology Code for Three Types of Classes

An additional element of type information is the superclass of a given class. The `extends` keyword is used in Java technology code to encode this type of class relationship. Figure B-11 shows an example of class inheritance.

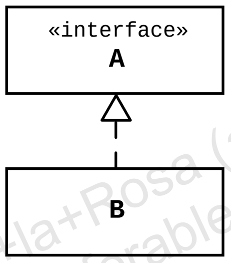


```
public class A {
  // members
}

public class B extends A {
  // members
}
```

Figure B-11 Java Technology Code for Class Inheritance

In Java technology, interfaces are implemented by classes. The `implements` keyword encodes this type of class relationship. Figure B-12 shows an example of interface implementation.



```
public interface A {
  // members
}

public class B implements A {
  // members
}
```

Figure B-12 Java Technology Code for Interface Implementation

Attributes

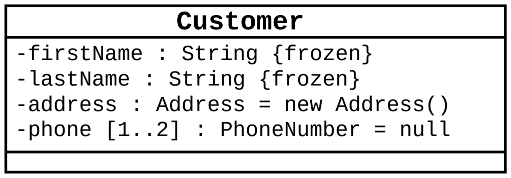
The UML syntax for an attribute declaration is:

```
[visibility] name [multiplicity] [: type] [= init-
value] [{property-string}]
```

The Java technology language syntax for an attribute declaration is:

```
[modifiers] type name [= init-value];
```

Figure B-13 illustrates an example class with attributes.


Customer
-firstName : String {frozen}
-lastName : String {frozen}
-address : Address = new Address()
-phone [1..2] : PhoneNumber = null

```
public class Customer {
  private final String firstName;
  private final String lastName;
  private Address address = new Address();
  private PhoneNumber[] phone = null;
}
```

Figure B-13 Java Technology Code for Attribute Declarations

The modifiers of an attribute declaration are determined by the following UML mapping rules:

- Visibility visibility maps to the attributes accessibility.

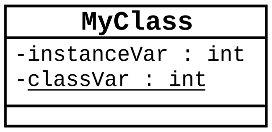
  In UML, visibility maps to the attributes accessibility. Table B-1 shows there are four predefined visibility values.

Table B-1 Accessibility Keywords

UML Symbol	Accessibility Keyword
+	`public`
#	`protected`
~	(no keyword)
-	`private`

- The `{frozen}` constraint maps to the `final` modifier.
- If the multiplicity of the attribute is greater than one, then the type of the attribute is either an array or a collection.
- Class scope can be declared.

An attribute can be declared to have class scope. In the UML, members (both attributes and methods) having class scope are represented by marking the member declaration with an underline. This maps to the `static` modifier. Figure B-14 illustrates an example.

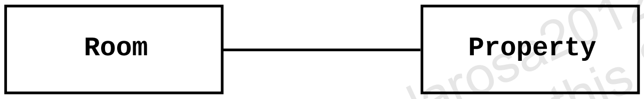


```
public class MyClass {
  private int instanceVar;
  private static int classVar;
}
```

Figure B-14 Java Technology Code for Class Scoped Attributes

Associations

Associations between classes are usually implemented as an attribute. Figure B-15 illustrates an example association.



```
public class Room {
  private Property property;
}
```

Figure B-15 Java Technology Code for an Association

There are several important features of associations:

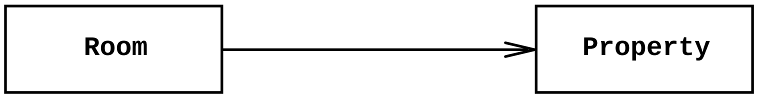
- Navigation
- Multiplicity
- Qualified associations
- Aggregation and composition

Navigation

The navigation arrow specifies which class must implement the association and its access method.

- Unidirectional

  A unidirectional association indicates that the association navigates in only one direction. Figure B-16 shows an example where the `Room` class navigates to the `Property` class; therefore, the `Room` class must include an attribute that maintains the association to an object of the Property class. Moreover, there is no attribute in the Property class to navigate to a Room object.



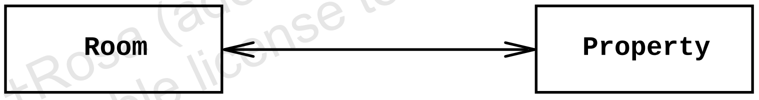
```
public class Room {
  private Property property;
}
```

```
public class Property {
  //members
}
```

Figure B-16 Java Technology Code for a Unidirectional Association

- Bidirectional

  A bidirectional association indicates that the association can navigate in both directions. Figure B-17 shows an example where the Room class navigates to the Property class and vice versa. Therefore, the Room class must include an attribute that maintains the association to an object of the Property class. Moreover, the Property class must include an attribute that maintains the association to an object of the Room class.



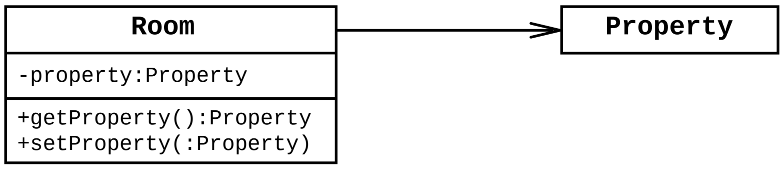
```
public class Room {
  private Property property;
}
```

```
public class Property {
  private Room room;
}
```

Figure B-17 Java Technology Code for a Bidirectional Association

Association Methods

Proper encapsulation also recommends keeping the association instance variable private and providing public access and mutator methods. In the case of a one-to-one association it is sufficient to supply simple accessor (get) and mutator (set) methods. Figure B-18 illustrates an example of this.

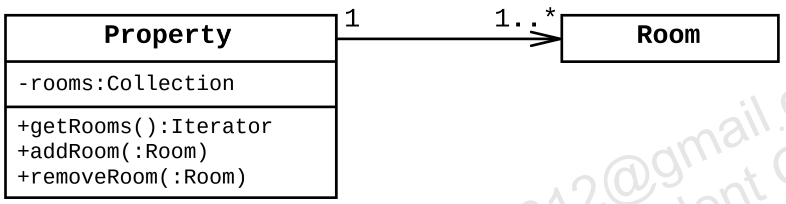


```
public class Room {
  private Property property;
  public Property getProperty() {
    return property;
  }
  public void setProperty(Property p) {
    property = p;
  }
}
```

Figure B-18 Java Technology Code for Association Methods

Association Multiplicity

When the multiplicity of an association is one-to-many or many-to-many, this adds significant complexity to the code that you must use to implement the association. There are many techniques for encoding these types of associations, such as using arrays. However, the most generic mechanism is to use the Java technology Collections API. For example, the `Property` class has a one-to-many association with the `Room` class. This can be represented with an attribute called `rooms` of type `Collection`. In this example, the implementation of that collection is a `HashSet`; you could also have used a list implementation class if the order of the rooms is important. Figure B-19 illustrates this example.



```
import java.util.*;
public class Property {
  private Collection rooms = new HashSet();
  public Iterator getRooms() {
    return rooms.iterator();
  }
  public void addRoom(Room r) {
    rooms.add(r);
  }
  public void removeRoom(Room r) {
    rooms.remove(r);
  }
}
```

Figure B-19 Java Technology Code for a One-to-Many Association

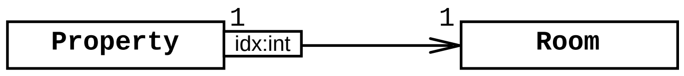
To access the collection of rooms in a property, you would need to encode a method to return an `Iterator` which enables you to iterate through each item in the collection. In Figure B-19, this is implemented by the `getRooms` method.

To add or remove rooms from the collection, you would need to encode two methods: `addRoom` and `removeRoom`. These methods control access to the association collection.

Qualified Associations

Qualified associations partition a one-to-many (or many-to-many) association into smaller groups. Typically, qualified associations identify a unique element in the collection.

A many-to-many association can be qualified by an integer index. Figure B-20 shows an example. In this example, the rooms of a property object are ordered by an index. A typical implementation would use a `List` attribute.

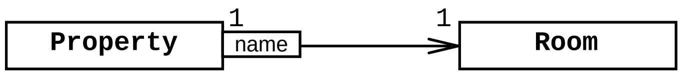


```
import java.util.*;
public class Property {
  private List rooms = new ArrayList();
  public Room getRoom(int idx) {
    return (Room) rooms.get(idx);
  }
  public void addRoom(int idx, Room r) {
    rooms.add(idx, r);
  }
  public void removeRoom(int idx) {
    rooms.remove(idx);
  }
}
```

Figure B-20 Java Technology Code for an Indexed Qualified Association

The accessor and mutator methods must permit the client of this class to access and manipulate the collection by the index. Therefore, the `getRoom` method takes an `idx` parameter to qualify which room is being requested. Similarly, the `addRoom` and `removeRoom` methods would use the index qualifier to specify which room is being added or removed.

A qualified association can also use a non-integer key to perform the lookup operation. This is sometimes called a “symbolic qualifier.” Typically, this qualifier is a string value. Figure B-21 shows an example. In this example, the rooms of a property are qualified by the name of the room. A Map data structure is usually used to implement a symbolic qualified association.



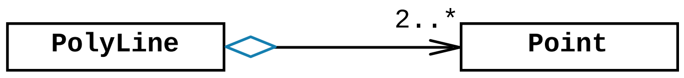
```
import java.util.*;
public class Property {
  private Map rooms = new HashMap();
  public Room getRoom(String name) {
    return (Room) rooms.get(name);
  }
  public void addRoom(Room r) {
    rooms.put(r.getName(), r);
  }
  public void removeRoom(Room r) {
    rooms.remove(r.getName());
  }
}
```

Figure B-21 Java Technology Code for a Symbolic Qualified Association

The accessor and mutator methods must permit the client of this class to access and manipulate the collection by the symbolic qualifier. Therefore, the getRoom method takes a name parameter to qualify which room is being requested. Similarly, the addRoom and removeRoom methods would use the name of the room object to specify which room is being added or removed.

Aggregation

An aggregation is an association that represents a whole-part relationship. The parts of an aggregation may be shared outside the aggregate class. Figure B-22 shows an example. In this example, a `PolyLine` object is constructed from two or more points.



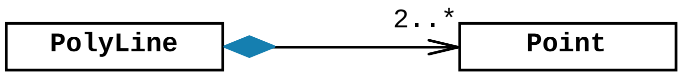
```
import java.util.*;
public class PolyLine {
  private List points = new LinkedList();
  public Iterator getPoints() {
    return points.iterator();
  }
  public void addPoint(Point p) {
    points.add(p);
  }
  public void removePoint(Point p) {
    points.remove(p);
  }
}
```

Figure B-22 Java Technology Code for an Aggregation

The `getPoints` accessor method enables the client class to iterate over each `Point` object. This exposes the `Point` objects to the client. This arrangement is permissible in an aggregation.

Composition

A composition is a specialization of an aggregation in which the custody of the part objects is completely owned by the whole object. To control this custody, the parts of a composition must not be exposed outside the composite class. Figure B-23 shows an example. In this example, the `Point` objects are not exposed to the client class.



```
import java.util.*;
public class PolyLine {
  private List points = new LinkedList();
  // no iterator method
  public void addPoint(int x, int y) {
    points.add(new Point(x,y));
  }
  public void removePoint(int x, int y) {
    DUMMY_POINT.setX(x);
    DUMMY_POINT.setY(y);
    points.remove(DUMMY_POINT);
  }
  private final static DUMMY_POINT = new Point(0,0);
}
```

Figure B-23 Java Technology Code for a Composition

To accomplish composition, there must not be any accessor method that returns the actual `Point` objects. In this example, no access method has been specified. The `addPoint` and `removePoint` methods are responsible for creating the internal `Point` objects. This convention requires that the parameters for these methods must include all of the necessary data to construct the `Point` objects (in this case the `x` and `y` coordinates of the point).

This type of control over the part objects in a composition can be extremely hard to accomplish. Typically, if the part objects are complex themselves, maintaining such tight control is too much of a burden. This is a design trade-off.

Summary

In this appendix, you were introduced to a few implementation considerations. Here are a few important concepts:

- It is important how components and classes are grouped into packages. You have seen a few principles that help guide that process.
- The UML can map closely with most OO languages. In this module, you have seen several common mappings between Class diagrams and Java technology code.

Appendix C

Testing the Software Solution

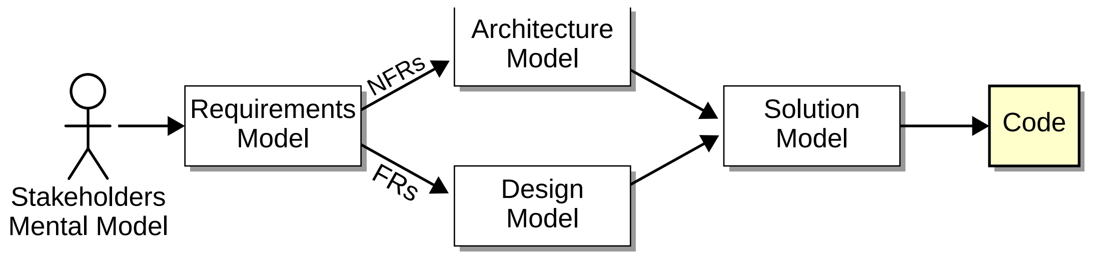
Objectives

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

- Define three major types of system tests
- Develop a functional test plan based on use cases

Process Map

The Testing workflow is a broad topic. This module introduces several areas of testing, but the focus will be on functional testing. Figure C-1 shows the activity and artifact discussed in this appendix.



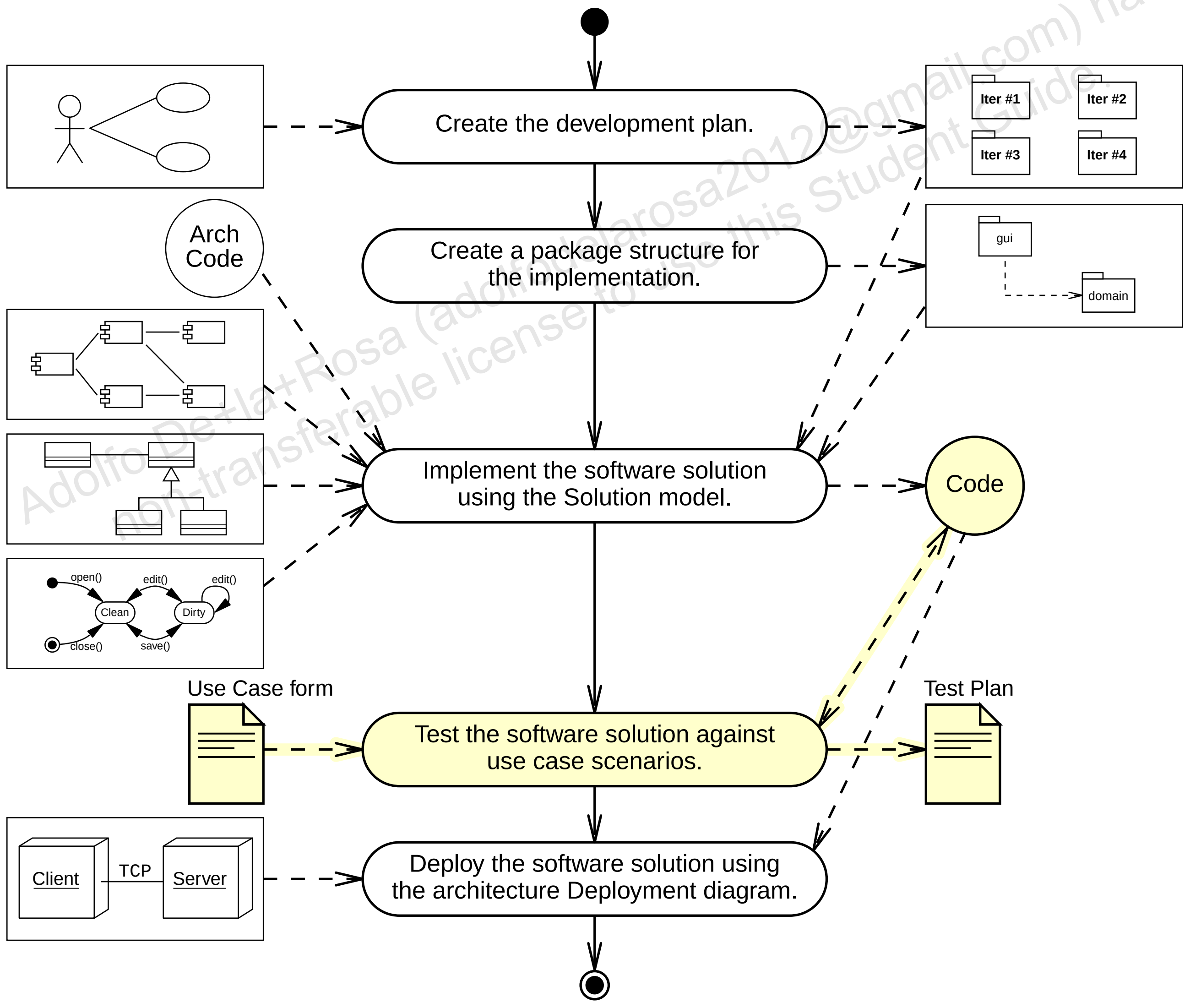

Create the development plan.
Iter #1
Iter #2
Iter #3
Iter #4
Arch Code
Create a package structure for the implementation.
gui
domain
Implement the software solution using the Solution model.
Code
open()
edit()
edit()
Clean
Dirty
close()
save()
Use Case form
Test Plan
Test the software solution against use case scenarios.
Client
TCP
Server
Deploy the software solution using the architecture Deployment diagram.

Figure C-1 Testing Workflow Process Map

Defining System Testing

The Test workflow is meant to “verify that the system correctly implements its specification.” (Jacobson, Booch, and Rumbaugh page 55) Ideally, testing is performed by a Software Tester; a job role that is independent of the development team. In practice, some testing is performed by developers. The reason why it is important to have independent testers is that the developer is often *too close* to the system being built. Developers are biased by their knowledge of how the system works so it is hard for them to be objective in finding ways of breaking the system. Also, at an unconscious level, most developers do not want to see their system fail.

There are numerous tests that could be performed. The following is a noncomprehensive list:

- Unit

  Unit testing tests individual components or classes.
- Integration

  Integration testing tests the correct functioning of groups of components; in particular, how major subsystems interact.
- Functional

  Functional testing tests the system as a whole against the use cases defined during the Requirements Gathering workflow.
- Regression

  Regressing testing is about going back and retesting things to make sure errors are fixed and that new errors did not occur. Regression testing can be done at any stage of development, but it is most important at the end of an iteration or at the end of a new release. Regressing testing can include unit, integration, functional tests, and so on.
- Load

  Load testing verifies that the system can handle the number of users and frequency of requests that are specified in the nonfunctional requirements. This test requires a complete system (even at increments) deployed on the specified hardware for production to verify that the measurements are realistic.

- Boundary condition

  Boundary condition testing verifies that the components, class, subsystems, and systems can handle inputs that fall on the boundary of acceptable values. For example, if a list is designed to contain no more than ten values, then you should test the addition of the tenth *and* eleventh item to the list. Adding the tenth item should succeed, but the eleventh should fail.

- Usability

  Usability testing verifies that the UI of the system is efficient, clear, correct, and user-friendly. Usability testing should be performed as early as possible. If a UI prototype is created, then this is the ideal time to perform usability tests.

The following subsections discuss the unit, integration, and functional types of testing in more detail.

Unit Tests

“A test written from the perspective of the programmer.” (Beck page 179)

Unit testing tests individual components or classes. Here are a few important concepts:

- Unit tests verify the behavior of individual components.

  For example, you might have an entity class that has a mutator method that takes a date string and stores that as a `Date` object. Then for this example, a good unit test is to pass in several different values including valid dates, strings that look like dates but are not valid (such as “29-February-2002”), and even strings that do not look like dates (such as “xyzzy”).
- Unit tests are written by developers.

  Some testing professionals would argue that developers should never write tests. However, most organizations cannot afford to devote a Software Tester to the complete life cycle of the project, so developers usually end up writing unit tests.

There are many techniques for creating unit tests. One technique is to add a `main` method in every class that needs unit testing. However, this technique is a violation of Separation of Concerns (that is, the functionality of the component and the testing of that functionality). A more robust approach is to use a testing framework.

Note – JUnit (`www.junit.org`) is an Open Source framework for testing Java technology components.

Integration Tests

“Integration testing test multiple components that have each received prior and separate unit testing.” (Whittaker page 77)

The focus of integration testing is to determine if large scale components or subsystems work together without error. Here are a few important concepts:

- Integration tests verify the behavior across multiple components, subsystems, and complete systems.

  System testing is a subset of Integration testing.
- Integration tests are usually written by testing professionals or by the development team.
- Integration tests should be built incrementally, just as the system is built incrementally.

  New integration tests are added for each increment. These new tests are added to the suite of regression tests for the complete system.

Functional Tests

“Most functional tests are written as black box tests working off a functional specification.” (Chillarege page 3)

Functional testing tests the system as a whole against the use cases defined during the Requirements Gathering phase. Here are a few important concepts:

- Functional tests verify the behavior of the (complete) system from the perspective of the user.

  Functional tests treat the system as a black box. This means that the internal behavior of the system is completely unknown, only the external inputs and outputs are seen. The use cases define this external behavior.
- Functional tests must be written by testing professionals in conjunction with the user community (if possible).
- Functional tests are based on the use cases of the system plus variations of all scenarios.

  Variations must test failures as well as successes to verify that the system handles failures properly.

Developing a Functional Test

A functional test is written based on a use case of the system. If the development team is using incremental development, then the functional tests can also be developed incrementally. This means that the functional tests for a specific use case are developed in the same iteration as the code for that use case.

The following are the fundamental steps involved in writing a functional test:

1. Identify the inputs for the test based on the inputs specified in the use case scenario.

   You need to create one functional test for every use case scenario. The best way to achieve this level of coverage is to use the use case scenarios. These scenarios will provide the inputs to the tests.

2. Specify the results of the test.

   You must identify the results of the test. There are usually two results to specify. The output of the system determines what appears on the screen or the form of printed reports. The persistent state of the system determines what has changed in the database after the test has been run.

3. Specify the conditions under which the test must be performed.

   You must specify the state of the persistent data store before the tests begin. What data must exist in order to execute the functional tests. For example, in the Hotel Reservation System, the database must include property and room data (and in some cases the DB should have existing reservations).

   Functional tests determine if the system behaves as specified in the requirements. Therefore, these tests are executed one at a time. However, these functional tests can also be used concurrently to see how well the system behaves for nonfunctional requirements. Therefore, functional test cases can be used for Load tests.

4. Write variations on this test to check failure processing, boundary conditions, and so on.

   You should attempt to define the boundary conditions of a specific use case scenario and create tests which verify that the system behaves correctly at the edges of these boundary conditions.

Identifying Functional Test Inputs

Functional test inputs include all data that must be entered into the system through the user interface.

For example, the following are the inputs for a single use case scenario for E1 Create a Reservation:

- The HotelApp is running with the Santa Cruz bed and breakfast.
- The begin and end dates of the reservation are January 31, 2002 and February 5, 2002, respectively.
- The type of room requested is double.
- The Blue room is selected from the room search.
- Jane Googol is entered into the customer search.
- A test credit card number is entered.

Identifying Functional Test Results

Functional test results include any output generated by the system as well as any persistent state change in the system.

For example, the follow are the results of the given use case scenario:

- A new reservation will exist for Jane Googol.
- The state of this reservation will be "confirmed."
- The reservation has an association with the Blue room (and only the Blue room).
- The payment entity for this reservation records the correct credit card number.

Identifying Functional Test Conditions

Functional test conditions include the existence of specific entities in the database and the existence of stubs for external systems and their data.

For example, the following are the conditions under which the given use case scenario must execute:

- The database contains the Santa Cruz bed and breakfast property and is associated with the Blue room, which is a double.
- Jane Googol is a valid customer in the database.
- A stub exists for the external credit card authorization system with data for test credit cards.
- No other concurrent tests must interact with that credit card account.

Creating Functional Test Variants

Tests variants are based on a use case scenario and explore the *state space* of the system.

For example, the following are test variants for the given use case scenario:

- Include a variant in which there are multiple customer entities for a single name; for example, John Smith.
- Include a variant in which a room search produces no available rooms.
- Include a variant in which the credit card is not authorized and the customer presents a second credit card.

Documenting Test Cases

The following are considerations for documenting functional test cases:

- All test cases should be stored in a Test Plan document.

  The Test Plan should be separate from the SRS.
- Each functional test must trace back to a use case in the SRS document.

  It is important to know that the functional tests were created to verify that the system supports the use cases. In other words, the test designers should not create arbitrary tests that the system was not designed to perform.

  However, it is important to create boundary condition variations on existing use case scenarios.
- If the system is large, then the Test Plan document can be split into separate documents—one for each major increment of the system.

- The Test Plan must be approved by the customer.

  The Test Plan document should be owned by the client-side stakeholders; although, it might be written by the System Tester in the development team.

- Other types of tests (such as load and usability tests) are usually documented separately.

  For large systems, you should separate the test plans and results in separate documents because these documents can become quite large. It is usually best to separate these documents by type of test: functional, integration, load, usability, and so on. However, another alternative is to group the tests by system increment.

Summary

In this appendix, you were introduced to Testing workflow and how to develop functional tests. Here are a few important concepts:

- Testing is critical to the success of a software project.
- Except for unit and integration testing, no tests should be developed and performed by the development team.
- Functional tests are developed from the system's use case scenarios.
- Functional test variants are created to test boundary conditions and system robustness.

Appendix D

Deploying the Software Solution

Objectives

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

- Explain system deployment
- Create an instance Deployment diagram

Process Map

This module describes a few activities of the Deployment workflow. Figure D-1 shows the activity and artifact discussed in this appendix.

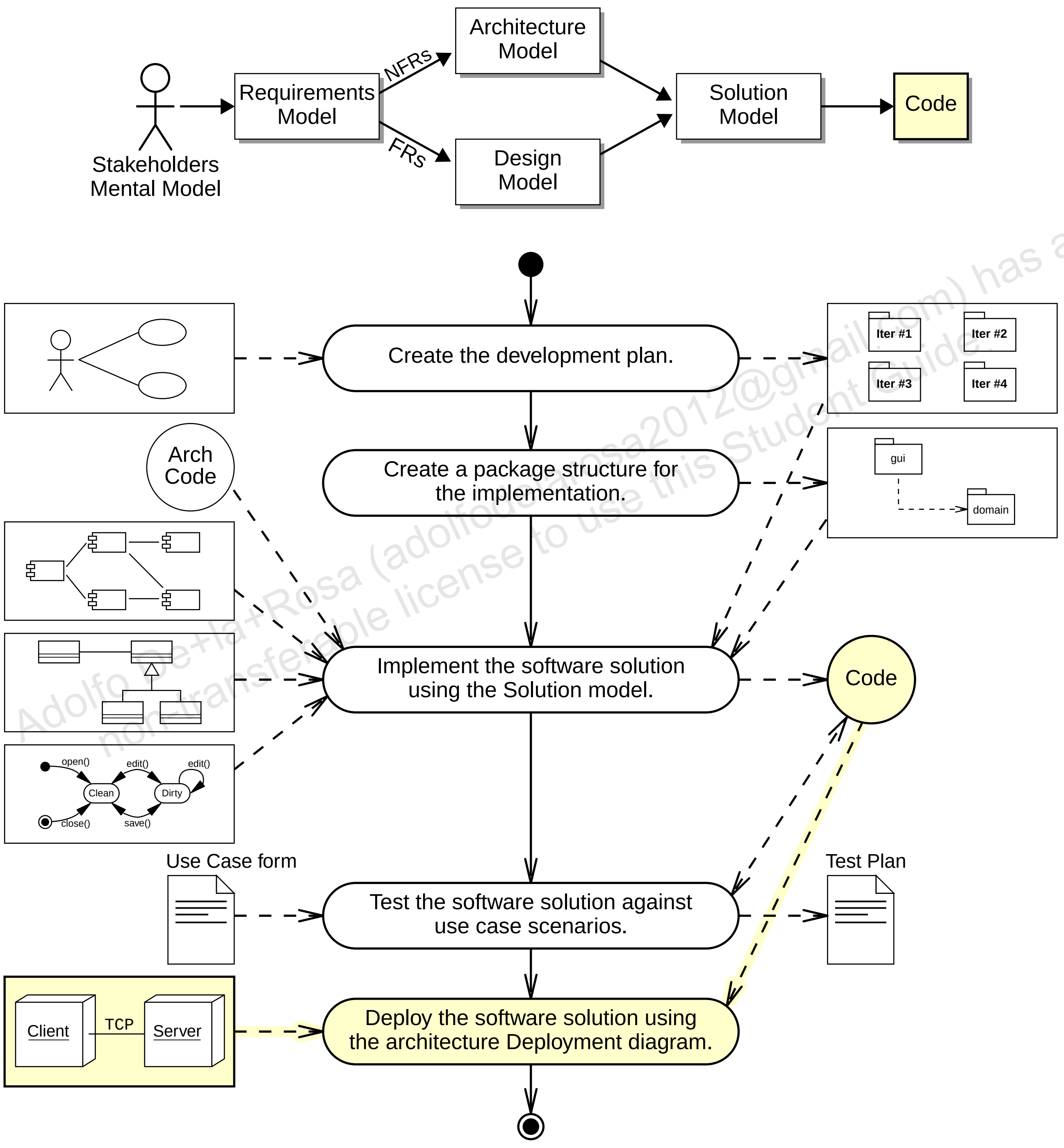

Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code
Create the development plan.
Iter #1
Iter #2
Iter #3
Iter #4
Arch Code
Create a package structure for the implementation.
gui
domain
Implement the software solution using the Solution model.
Code
open()
edit()
edit()
Clean
Dirty
close()
save()
Use Case form
Test Plan
Test the software solution against use case scenarios.
Client
TCP
Server
Deploy the software solution using the architecture Deployment diagram.

Figure D-1 Deployment Workflow Process Map

Explaining System Deployment

The purpose of the Deployment workflow is to place the software system into the production environment and to make sure that all users have been properly trained to use the system. Relative to the Unified Process, the Deployment workflow occurs in the Transition phase, but some of these activities can be performed in the Construction phase.

Deployment of the software solution is usually performed by the client company's system administration team. The development team usually only has a consulting role.

The following are a few of the many possible deployment activities:

- Purchase necessary hardware and software.

  The Deployment diagram and the Tiers/Layers diagram record the necessary hardware machines and software packages that need to be acquired to construct the production environment. The descriptive Deployment diagram must be transformed into an instance Deployment diagram which specifies the hardware constraints for each machine in the system.

- Construct the networking infrastructure.

  Buy and connect the networking hardware to construct the topology specified in the Deployment diagram. This might require buying additional hardware and software for routers, firewalls, and so on.

- Install the third-party and custom software.

  Using the architecture tiers and layers Package diagram as a guide, install all necessary software on the production hardware.

- Migrate legacy data to new system (if necessary).

  Convert any legacy data of the old system into the new system. This usually requires a set of scripts or programs for performing this conversion. This can be a complex task, so it must be scheduled during the Construction and Transition phases.

- Test the installation.

  Perform all tests (functional and nonfunctional) on the production environment using the test databases instead of the production databases.

- Train the staff.

  Create all necessary documentation and train the users on the new system. This activity is extremely important and should be performed well in advance of the *go live* date.

- Go live.

  It is important to celebrate the creation of the system. Many organizations throw a party for the development team (as well as the client-side stakeholders). This is not frivolous because it boosts the morale of the development team and builds better rapport between the client-side stakeholders and the development team for future projects.

Using the Deployment Diagram

The following describe how to use the Deployment diagram in the Deployment workflow:

- The Deployment diagram of the system is the architect's vision of how the system should be deployed.

  The Deployment diagram specifies the production environment as specified by the Architect. This vision might need to be altered to integrate with the client's existing production environment.

- The Deployment diagram should provide both the structure of the network and the set of software components.

  It is important to note that the Deployment diagram is probably not comprehensive. The Deployment Specialists should be able to take the Deployment diagram and determine the missing pieces, such as additional networking hardware.

- The "descriptive" Deployment diagram (created in the Elaboration phase) needs to be elaborated with details in the "instance" Deployment diagram before deployment can proceed.

  The instance Deployment diagram shows the specifications for the hardware nodes in the descriptive Deployment diagram. This will guide the client's purchases or upgrades.

Creating an Instance Deployment Diagram

Creating an instance Deployment diagram from a descriptive Deployment diagram requires adding the following features:

- Detailed network configuration

  The descriptive Deployment diagram only shows the links (network connections) between the major hardware nodes. In the instance Deployment diagram you should also show the additional networking hardware for the production environment.

- Detailed hardware configuration

  In the instance Deployment diagram you should specify the hardware constraints (number of CPUs, clock speed, memory (RAM) and storage (disks) requirements) for each piece of hardware.

- Detailed software configuration

  In the instance Deployment diagram you should specify the major software components, such as JAR files, application files, scripts, and other data files.

Figure D-2 shows the descriptive Deployment diagram for the Hotel Reservation System.

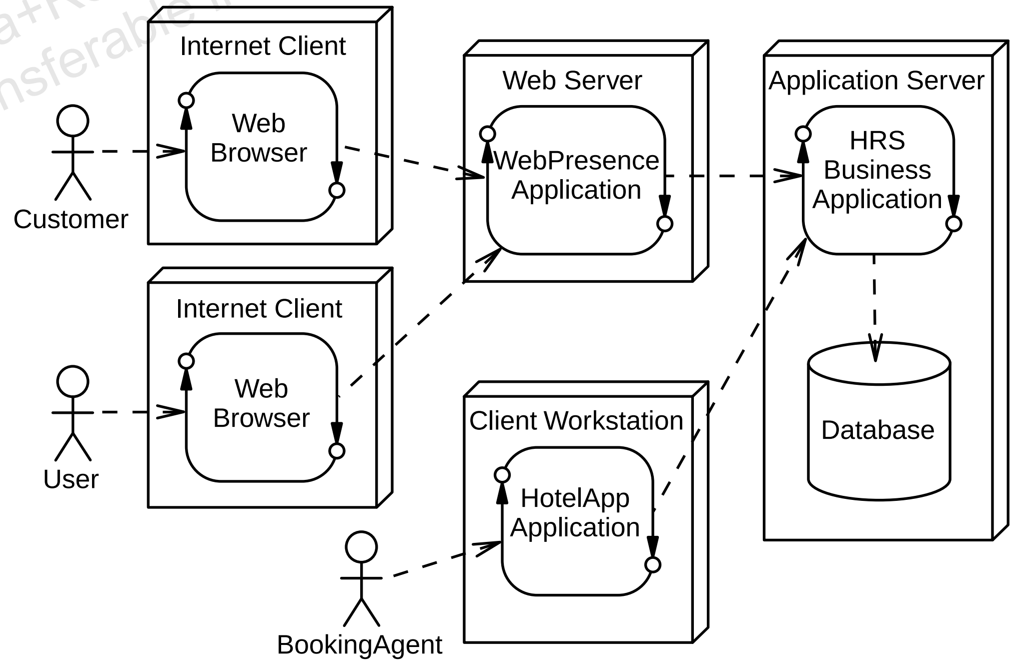


Figure D-2 Hotel Reservation Descriptive Deployment Diagram

The following subsections describe the instance Deployment diagrams for each major hardware component.

Kiosk Application Instance Deployment Diagram

The KioskApp is an internal web server. These are a few requirements on the kiosk machines:

- One kiosk in the lobby of each property.
- The kiosk does not include a connection to the Internet.

The KioskApp is very basic and does not require expensive hardware, except that the monitor must be a touch-sensitive. Figure D-3 illustrates the hardware constraints and software allocations.

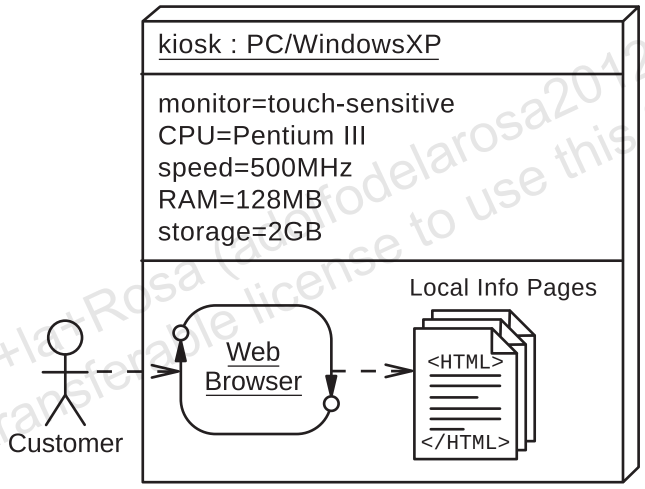


Figure D-3 Instance Deployment Diagram for the KioskApp

WebPresenceApp Instance Deployment Diagram

The WebPresenceApp is an external web application server. The Apache web server software will be used to server the static Web site pages and the Tomcat Web container will be used to server the dynamic elements of the Bay View system. The WebPresenceApp software requires the services interfaces and RMI stub classes, which are stored in the `services.jar` file. Figure D-4 illustrates this configuration.

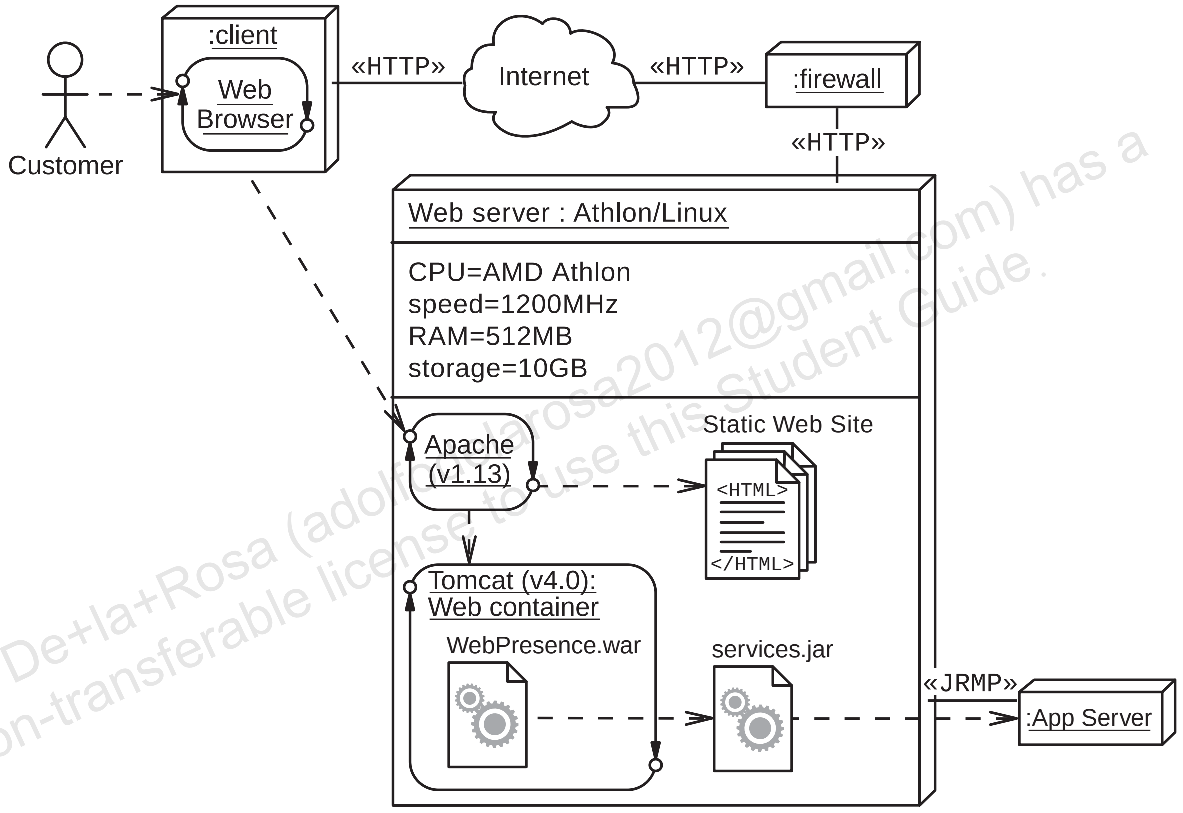


Figure D-4 The Instance Deployment Diagram for the WebPresenceApp

HotelApp Instance Deployment Diagram

The KioskApp is an internal application. These are a few requirements on the user workstations:

- HotelApp clients exist at each hotel location.
- HotelApp communicates to the server over a Virtual Private Network (VPN) using the RMI protocol (JRMP).

  Because the hotels are distributed over a great distance, these workstations must be connected to the Internet to communicate to the remote business tier (at the Santa Cruz facility). Because the system will be sending confidential data, the Deployment Specialist has decided to use VPN software to secure the communication path through the Internet.

Figure D-5 illustrates this configuration.

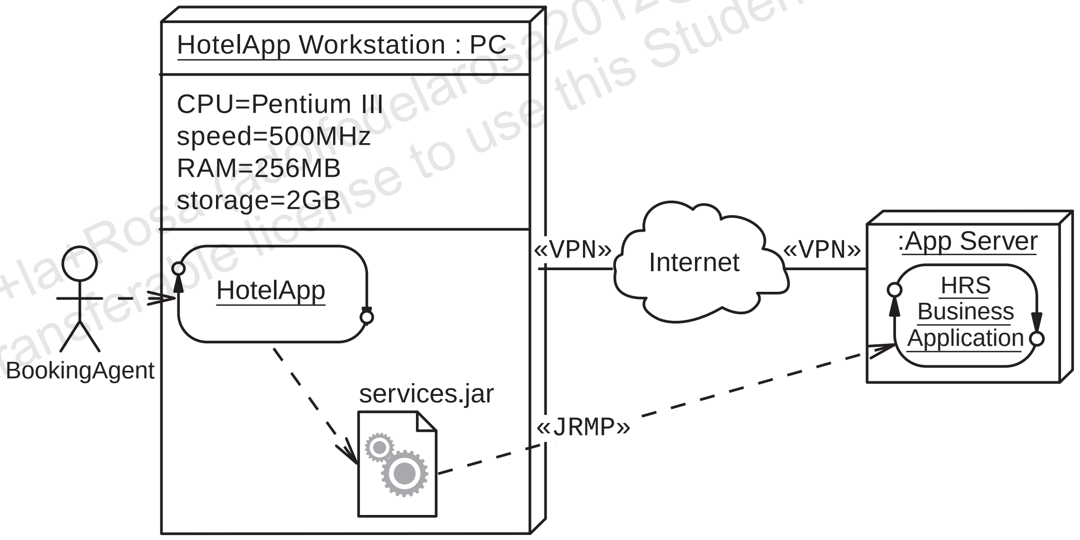


Figure D-5 The Instance Deployment Diagram for the HotelApp

Remote Application Server Instance Deployment Diagram

The remote application server runs the business services and the database management system. The HRS Business Application acts as the remote services implementation; therefore, this application binds the service objects to names in the RMI registry. This application requires the actual implementation classes of the services. The implementation is stored in the `services_impl.jar` file. This application also requires access to the PostgreSQL database software. Figure D-6 illustrates this configuration.

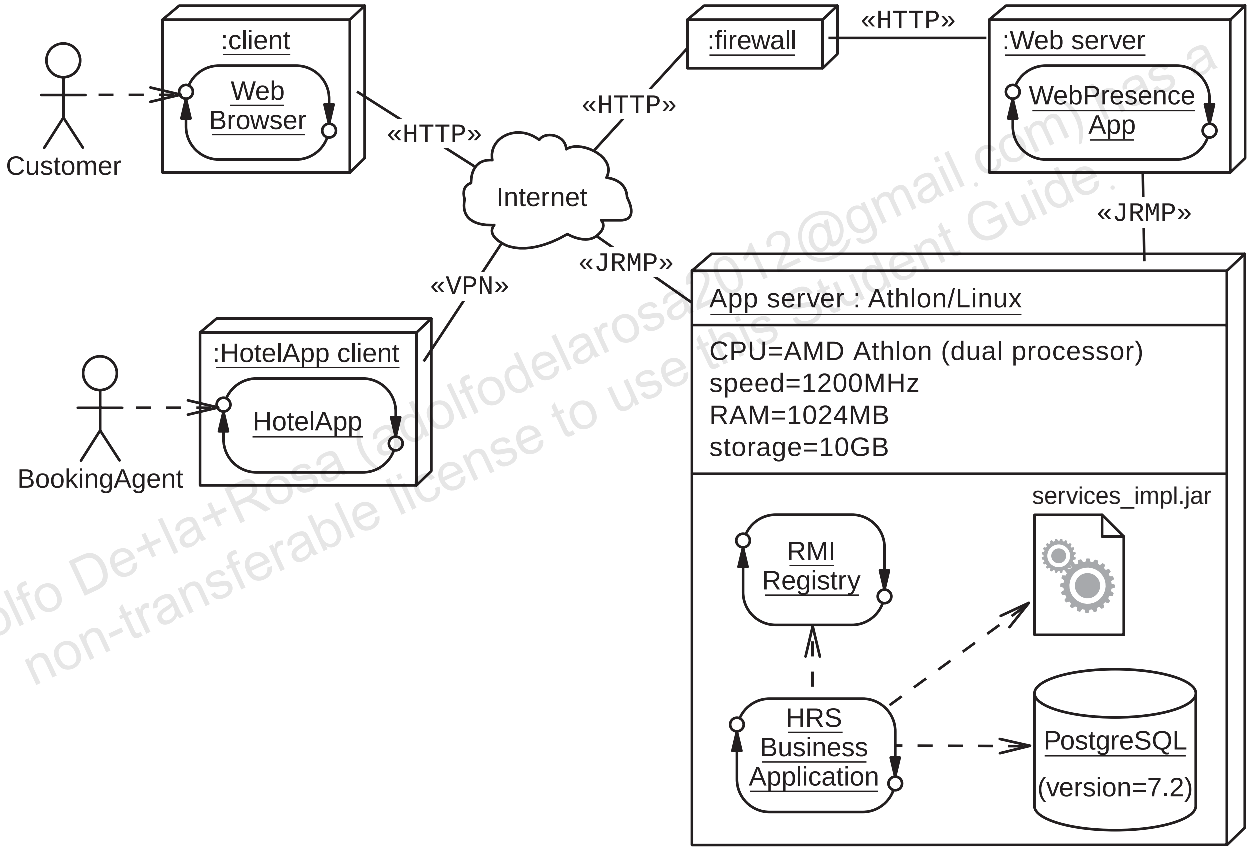


Figure D-6 Instance Deployment Diagram for the Business Tier Host

Summary

In this appendix, you were introduced to the Deployment workflow. Here are a few important concepts:

- The Deployment workflow takes the system implementation into production.
- The Deployment diagram specifies how the system is to be deployed in the production configuration.
- An instance Deployment diagram elaborates the descriptor Deployment diagram by clearly specifying the configuration of:
  - Networks
  - Machines
  - Software components

Appendix E

Quick Reference for UML

This appendix is designed to serve as a quick reference for the Unified Modeling Language (UML) version 1.4, with some additional UML version 2.2 references.

Additional Resources

Additional resources – The following references provide additional details on the topics described in this appendix:

- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Folwer, Martin, with Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- 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 – Additional UML resources are available online at: `http://www.omg.org/uml/`.

UML Basics

Unified Modeling Language (UML) is a graphical language for modeling software systems. UML is not a programming language, it is a set of diagrams that can be used to specify, construct, visualize, and document software designs. Software engineers use UML diagrams to construct and explain their software designs just as a building architect uses blueprints to construct and explain their building designs. UML has diagrams to assist in every part of the application development process from requirements gathering through design, and into coding, testing, and deployment.

UML was developed in the early 1990's by three leaders in the object modeling world: Grady Booch, James Rumbaugh, and Ivar Jacobson. Their goal was to unify the major methods that they had previously developed to create a new standard for software modelling. UML is now the most commonly used modeling language. UML is currently maintained by the Object Management Group (OMG). The UML specification is available on the OMG web site at `http://www.omg.org/uml/`.

The UML is not a process for how to do analysis and design. UML is only a set of tools to use in a process. UML is frequently used with a process such as the Unified Software Development Process. Sun Microsystems OO-226, *Object Oriented Analysis and Design Using UML*, is a five day course that teaches effective methods of analysis and design using UML language, the USDP method, and software patterns.

UML defines nine standard types of diagrams. Table E-1 provides a list of these diagrams, an informal description, and best-use recommendations.

Table E-1 Types of UML Diagrams

Diagram Name	**Description**	**Best Use**
Use Case	A Use Case diagram is a simple diagram that shows who is using your system and what processes they will perform in the system.	The Use Case diagram is an extremely important diagram for the Requirements Gathering and Analysis workflows. Throughout the entire development, all work should be traceable back to the Use Case diagram.

Table E-1 Types of UML Diagrams (Continued)

Diagram Name	Description	Best Use
Class	A Class diagram shows a set of classes in the system and the associations and inheritance relationships between the classes. Class nodes might also contain a listing of attributes and operations.	The Class diagram is essential for showing the structure of the system and what needs to be programmed. Most UML case tools can generate code based on the Class diagram.
Object	An Object diagram shows specific object instances and the links between them. An Object diagram represents a snapshot of the system objects at a specific point in time.	The Object diagram can be used to clarify or validate the Class diagram.
Activity	An Activity diagram is essentially a flow chart with new symbols. This diagram represents the flow of activities in a process or algorithm.	Even in an OO system, it can sometimes be useful to consider processes without thinking in terms of objects. Activity diagrams are especially useful for modeling real world business systems during the Requirements Gathering workflow.
Communication	The Sequence diagram and the Communication diagram both show processes from an object oriented perspective. The main difference is that the layout of the Communication diagram puts more focus on the objects rather than the sequence. The Communication diagram was formerly called Collaboration diagram.	Sequence diagrams are typically easier to read than Communication diagrams. Many people prefer to only use Sequence diagrams. A Communication diagram might be preferable if you want more focus on the objects than the sequence.
Sequence	Sequence diagrams show a process from an object oriented perspective by showing how a process is executed by a set of objects or actors.	The Sequence diagram is essential for assigning responsibilities to classes by considering how they can work together to implement the processes in the system.

Table E-1 Types of UML Diagrams (Continued)

Diagram Name	Description	Best Use
State Machine	A State Machine diagram shows how a particular object changes behavioral state as various events happen to it.	The State Machine diagram is very useful in understanding objects that change behavioral states in significant ways.
Component	A Component diagram shows the major software components of a system and how they can be integrated. Component diagrams can contain non-OO software components such as legacy procedural code and web documents.	Component diagrams can be a good way to show how all of the OO and non-OO components fit together in your system. It is also a good way to look at the high-level software structure of your system.
Deployment	A Deployment diagram shows the hardware nodes in the system.	The Deployment diagram is useful for seeing how a distributed system will be configured. Software components might be displayed inside the hardware nodes to show how they will be deployed.

Note – UML 2.2 has five additional diagrams: Package, Timing, Interaction Overview, Composite Structure, and Profile diagram.

General Elements

In general, UML diagrams represent concepts, depicted as symbols (also called nodes), and relationships among concepts, depicted as paths (also called links) connecting the symbols. These nodes and links are specialized for the particular diagram. For example, in Class diagrams, the nodes represent object classes and the links represent associations between classes and generalization (inheritance) relationships.

There are other elements that augment these diagrams, including: packages, stereotypes, annotations, constraints, and tagged values.

Packages

In UML, packages enable you to arrange your modeling elements into groups. UML packages are a generic grouping mechanism and should not be directly associated with Java technology packages. However, you can use UML packages to model Java technology packages. Figure E-1 demonstrates a package diagram that contains a group of classes in a Class diagram.

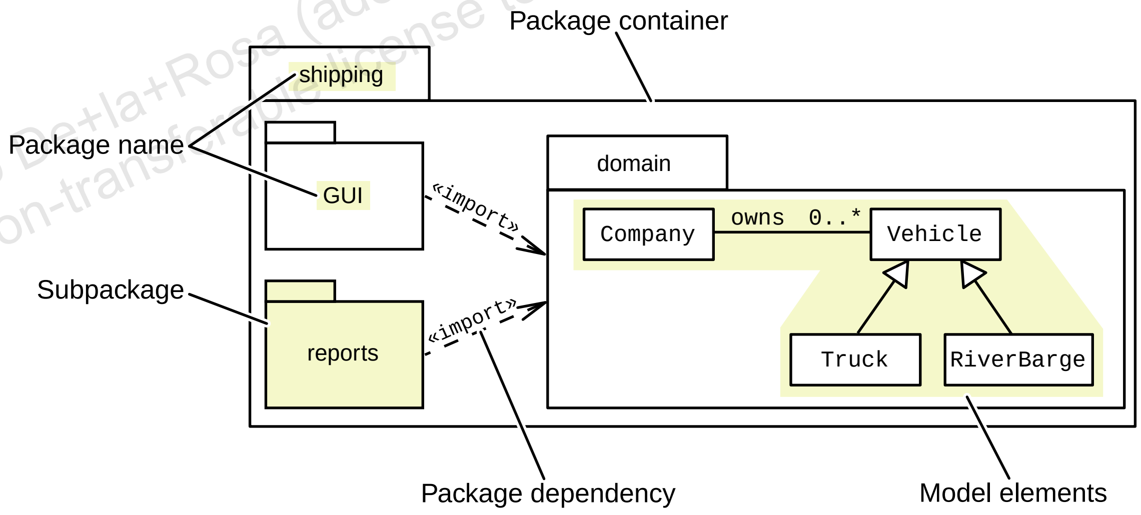


Figure E-1 Example Package

Mapping to Java Technology Packages

The mapping of UML packages to Java technology packages implies that the classes would contain the package declaration of package `shipping.domain`. For example, in the file `Vehicle.java`:

```
package shipping.domain;

public class Vehicle {
  // declarations
}
```

Figure E-1 on page E-6 also demonstrates a simple hierarchy of packages. The `shipping` package contains the `GUI`, `reports`, and `domain` `subpackages`. The dashed arrow from one package to another indicates that the package at the tail of the arrow uses (imports) elements in the package at the head of the arrow. For example, `reports` uses elements in the `domain` package as follows:

```
package shipping.reports;

import shipping.domain.*;

public class VehicleCapacityReport {
  // declarations
}
```

Note – In Figure E-1 on page E-6, the `shipping.GUI` and `shipping.reports` packages have their names in the body of the package box rather than in the head of the package box. This is done only when the diagram does not expose any of the elements in that package.

Stereotypes

The designers of UML understood that they could not build a modeling language that would satisfy every programming language and every modeling need. They built several mechanisms into UML to enable modelers to create their own semantics for model elements (nodes and links). Figure E-2 shows the use of a stereotype tag «interface» to declare that the class node Set is a Java technology interface declaration. Stereotype tags can adorn relationships as well as nodes. There are over a hundred standard stereotypes. You can create your own stereotypes to model your own semantics.

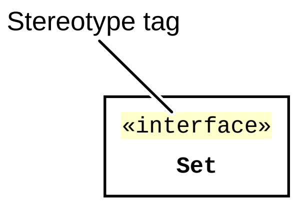


Figure E-2 Example Stereotype

Annotation

The designers of UML also built a method for annotating the diagrams into the UML language. Figure E-3 shows a simple annotation.

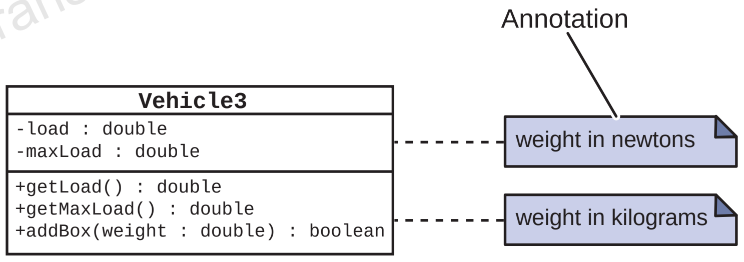


Figure E-3 Example Annotation

Annotations can contain notes about the diagram as a whole, notes about a particular node or link, or even notes about an element within a node. The dotted link from the annotation points to the element being annotated. If there is no link from the annotation, then the note is about the whole diagram.

Constraints

Constraints enable you to model certain conditions that apply to a node or link. Figure E-4 shows several constraints. The topmost constraint specifies that the `Player` objects must be stored in a persistent database. The middle constraint specifies that the captain and co-captain must also be members of the team's roster. The constraint on the bottom specifies the minimum number of players by gender.

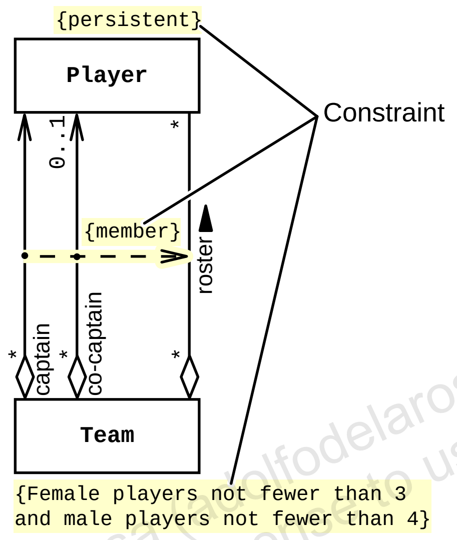


Figure E-4 Example Constraints

Tagged Values

Figure E-5 shows several examples of how tagged values enable you to add new properties to nodes in a diagram.

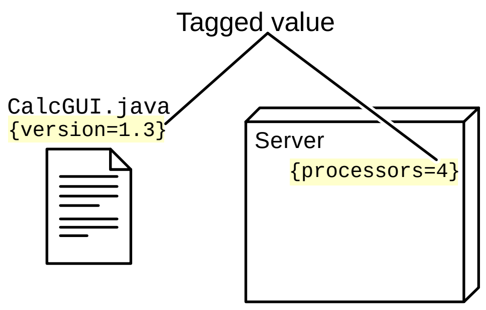


Figure E-5 Example Tagged Values

Use Case Diagrams

A Use Case diagram represents the functionality provided by the system to external users. The Use Case diagram is composed of actors, use case nodes, and their relationships. Actors can be humans or other systems.

Figure E-6 shows a simple banking Use Case diagram. An actor node can be denoted as a *stick figure* (as in the three Customer actors) or as a class node (see "Class Nodes") with the stereotype of «actor». There can be a hierarchy of actors.

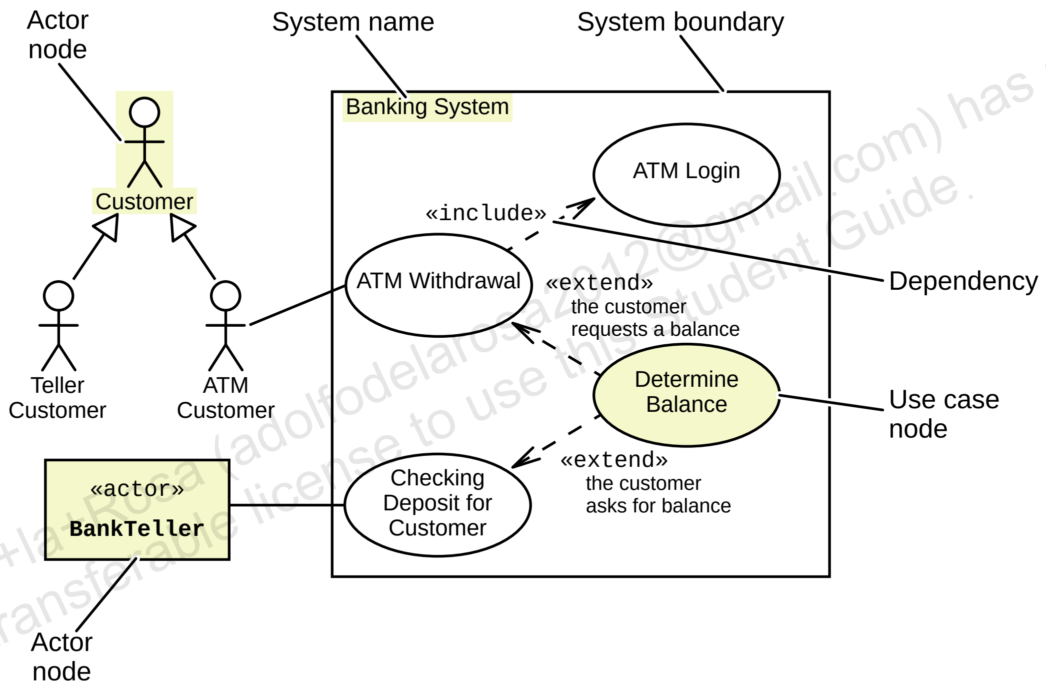


Figure E-6 An Example Use Case Diagram

A use case node is denoted by a labeled oval. The label indicates the activity summary that the system performs for the actor. Use case nodes are grouped into a system box, which is usually labeled in the top left corner. The relationship "actor uses the system to" is represented by the solid line from the actor to the use case node.

Use case nodes can depend on other use cases. For example, the "ATM Withdrawl" use case uses the "ATM Login" use case. Use case nodes can also extend other use cases to provide optional functionality. For example, the "Determine Balance" use case can be used to extend the "Checking Deposit for Customer" use case.

Class Diagrams

A Class diagram represents the static structure of a system. These diagrams are composed of classes and their relationships.

Class Nodes

Figure E-7 shows several *class nodes*. You do not have to model every aspect of an class every time that class is used in a diagram.

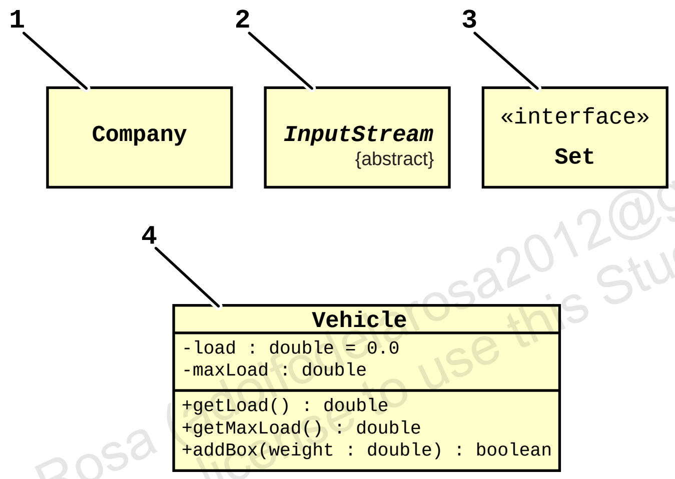


Figure E-7 Several Class Nodes

A class node can just be the name of the class, as in Examples 1, 2, and 3 of the figure. Example 1 is a concrete class, where no members are modeled. Example 2 is an abstract class (name is in italics). Example 3 is an interface. Example 4 is a concrete class, where members are modeled

Figure E-8 illustrates the element of a class node.

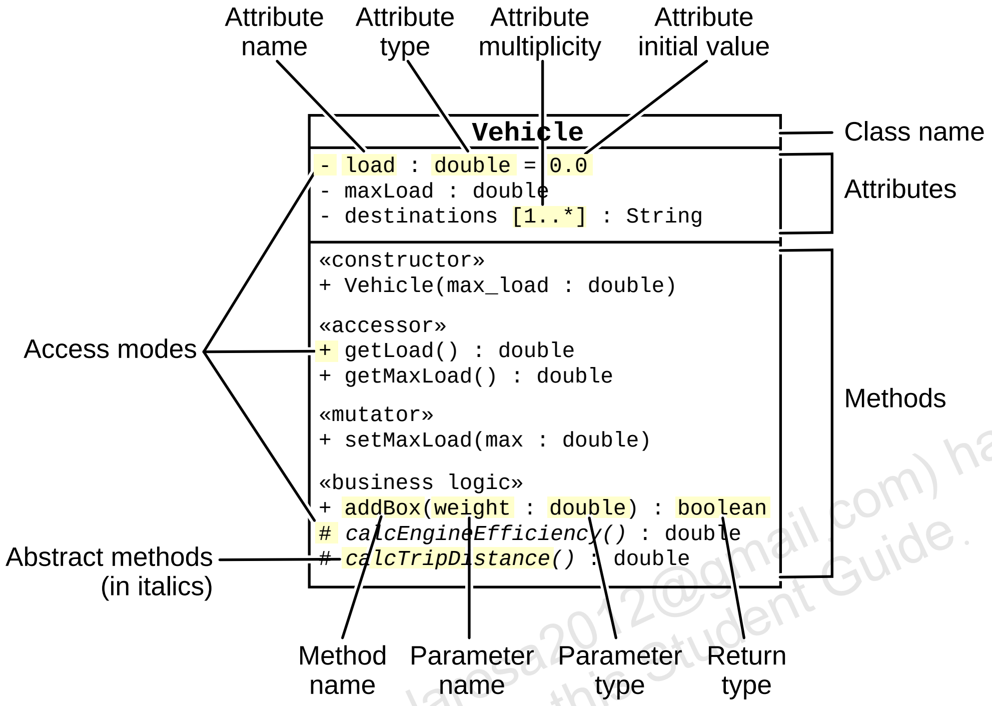


Figure E-8 Elements of a Class Node

A fully specified class node has three basic compartments:

- The name of the class in the top
- The set of attributes under the first bar

  An attribute is specified by five elements, including access mode, name, multiplicity, data type, and initial value. With attributes, all of the elements are optional except for the name element.
- The set of methods under the second bar

  A method is specified by four elements, including access mode, name, parameter list (a comma-delimited list of parameter name and type), and the return type. With methods, all but the name are optional, except for the name element. If the return value is not specified, then no value is returned (`void`). The name of an abstract method is marked in italics.

You can use stereotypes to group attributes or methods together. For example, you can separate accessor, mutator, and business logic methods from each other for clarity. And because there is no UML-specific notation for constructors, you can use the «`constructor`» stereotype to mark the constructors in your method compartment.

Table E-2 shows the valid UML access mode symbols.

Table E-2 UML Defined Access Modes and Their Symbols

Access Mode	Symbol
private	-
package private	~
protected	#
public	+

Figure E-9 shows an example class node with elements that have class (or static) scope. This is denoted by the underline under the element. For example, `counter` is a static data attribute and `getTotalCount` is a static method.

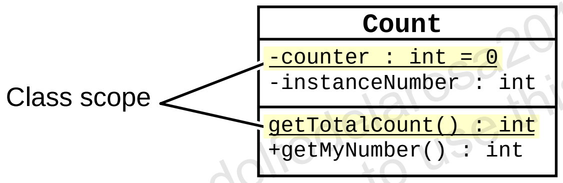


Figure E-9 An Example Class Node With Static Elements

You could write the `Count` class in the Java programming language as:

```
public class Count {
  private static int counter = 0;
  private int instanceNumber;

  public static int getTotalCount() {
    return counter;
  }
  public int getMyNumber() {
    return instanceNumber;
  }
}
```

Inheritance

Figure E-10 shows class inheritance through the generalization relationship arrow.

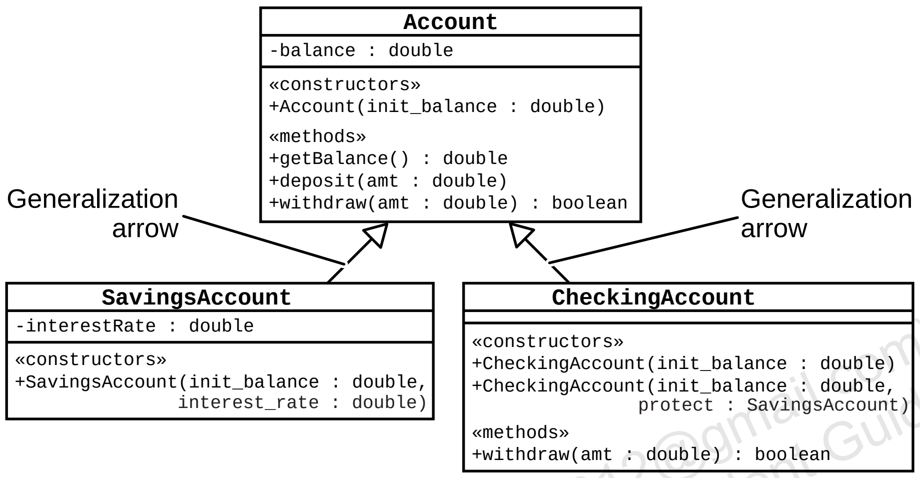

Account
-balance : double
«constructors»
+Account(init_balance : double)
«methods»
+getBalance() : double
+deposit(amt : double)
+withdraw(amt : double) : boolean
Generalization arrow
Generalization arrow
SavingsAccount
-interestRate : double
«constructors»
+SavingsAccount(init_balance : double, interest_rate : double)
CheckingAccount
«constructors»
+CheckingAccount(init_balance : double)
+CheckingAccount(init_balance : double, protect : SavingsAccount)
«methods»
+withdraw(amt : double) : boolean

Figure E-10 Class Inheritance Relationship

Class inheritance is implemented in the Java programming language with the `extends` keyword. For example:

```
public class Account {
  // members
}

public class SavingsAccount extends Account {
  // members
}

public class CheckingAccount extends Account {
  // members
}
```

Interface Implementation

Figure E-11 shows how to use the "realization" arrow to model a class that is implementing an interface.

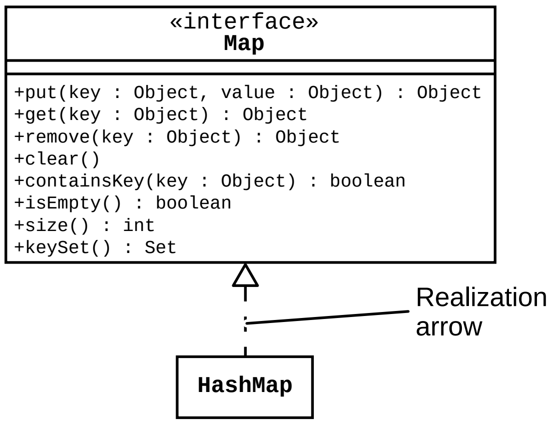

«interface»
Map
+put(key : Object, value : Object) : Object
+get(key : Object) : Object
+remove(key : Object) : Object
+clear()
+containsKey(key : Object) : boolean
+isEmpty() : boolean
+size() : int
+keySet() : Set
Realization arrow
HashMap

Figure E-11 An Example of a Class Implementing an Interface

An interface is implemented in the Java programming language with the `implements` keyword. For example:

```
public interface Map {
  // declaration here
}

public class HashMap implements Map {
  // definitions here
}
```

Association, Roles, and Multiplicity

Figure E-12 shows an example association. An *association* is a link between two types of objects and implies the ability for the code to navigate from one object to another.

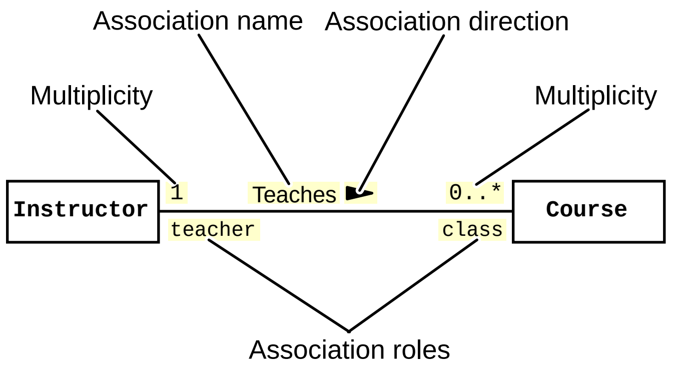


Figure E-12 Class Associations and Roles

In this diagram, "Teaches" is the name of the association with a directional arrow pointing to the right. This association can be read as "an instructor teaches a course." You can also attach roles to each end of the association. In the figure, the "teacher" role indicates that the instructor is the teacher for a given course. All of these elements are optional if the association is obvious.

This example also demonstrates how many objects are involved in each role of the association. This is called *multiplicity*. In this example, there is only one teacher for every class, which is denoted by the "`1`" on the `Instructor` side of the association. Also any given teacher might teach zero or more courses, which is denoted by the "`0..*`" on the `Course` side. You can leave out the multiplicity for a given role if it is always one. You can also abbreviate "zero or more" as just "`*`".

You can express multiplicity values as follows:

- A range of values – For example, `2..10` means "at least 2 and up to 10
- A disjoint set of values – For example, `2,4,6,8,10`
- A disjoint set of values or ranges – For example, `1..3,6,8..11`

However, the most common values for multiplicity are exactly one (`1` or left blank), zero or one (`0..1`), zero or more (`*`), or one or more (`1..*`).

Associations are typically represented in the Java programming language as an attribute in the class at the tail of the relationship (specified by the direction indicator). If the multiplicity is greater than one, then some sort of collection or array is necessary to hold the elements.

Also, in Figure E-12 on page E-16 the association between an instructor and a course might be represented in the `Instructor` class as:

```
public class Instructor {
  private Course[]  classes = new Course[MAXIMUM];
}
```

or as:

```
public class Instructor {
  private List  classes = new ArrayList();
}
```

The latter representation is preferable if you do not know the maximum number of courses any given instructor might teach.

Aggregation and Composition

An *aggregation* is an association in which one object contains a group of parts that make up the *whole* object (see Figure E-13). For example, a car is an aggregation of an engine, wheels, body, and frame. Composition is a specialization of aggregation in which the parts cannot exist independently of the *whole* object. For example, a human is a composition of a head, two arms, two legs, and a torso. If you remove any of these parts without surgical intervention, the whole person is going to die.

In the example in Figure E-13, a sports league, defined as a sports event that occurs seasonally every year, is a composition of divisions and schedules. A division is an aggregation of teams. A team might exist independently of a particular season; in other words, a team might exist for several seasons (leagues) in a row. Therefore, a team might still exist even if a division is eliminated. Moreover, a game can only exist in the context of a particular schedule of a particular league.

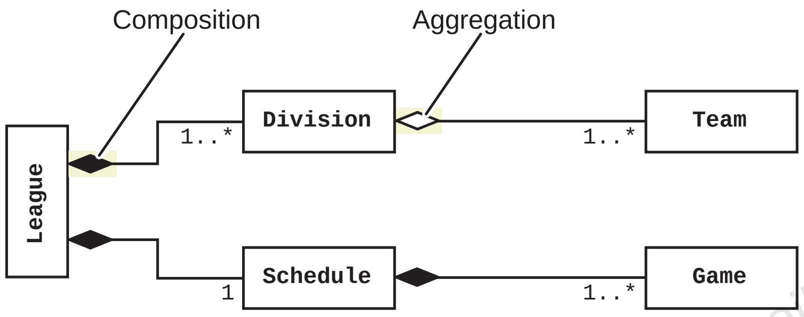


Figure E-13 Example Aggregation and Composition

Association Classes

An association between two classes might have properties and behavior of its own. Association classes are used to model this characteristic. For example, players might be required to register for a particular division within a sports league, as Figure E-14 shows. The association class is attached to the association link by a dashed line.

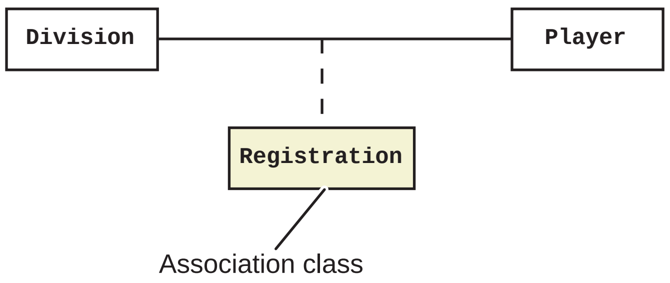


Figure E-14 A Simple Association Class

Figure E-15 shows an association class that is used by two associations and that has two private attributes. This example indicates that a `Game` object is associated with two opposing teams and each team will have a score and a flag specifying whether that team forfeited the game.

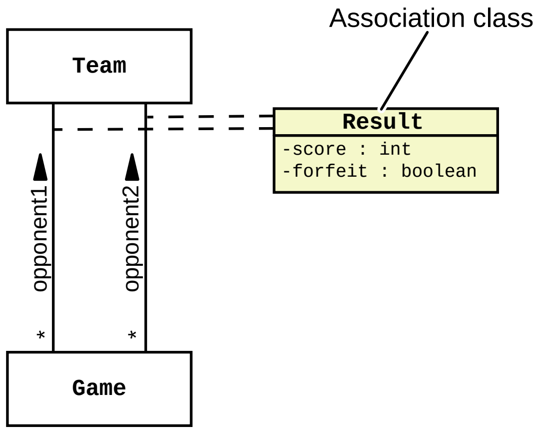


Figure E-15 A More Complex Association Class

You can use the Java programming language to represent an association class in several different ways. One way is to code the association class as a standard Java technology class. For example, registration could be coded as follows:

```
public class Registration {
  private Division division;
  private Player player;
  // registration data
  // methods...
}
public class Division {
  // data
  public Registration retrieveRegistration(Player p) {
    // lookup registration info for player
    return new Registration(this, p, ...);
  }
  // methods...
}
```

Another technique is to code the association class attributes directly into one of the associated classes. For example, the `Game` class might include the score information as follows:

```
public class Game {
  // first opponent and score details
  private Team opponent1;
```

```
    private int opponent1_score;
    private boolean opponent1_forfeit;
    // second opponent and score details
    private Team opponent2;
    private int opponent2_score;
    private boolean opponent2_forfeit;
    // methods...
  }
```

Other Association Elements

There are several other parts of associations. This section presents the constraints and qualifiers elements.

An association constraint enables you to augment the semantics of two or more associations by attaching a dependency arrow between them and tagging that dependency with a constraint. For example in Figure E-16, the captain and co-captain of a team are also members of the team's roster.

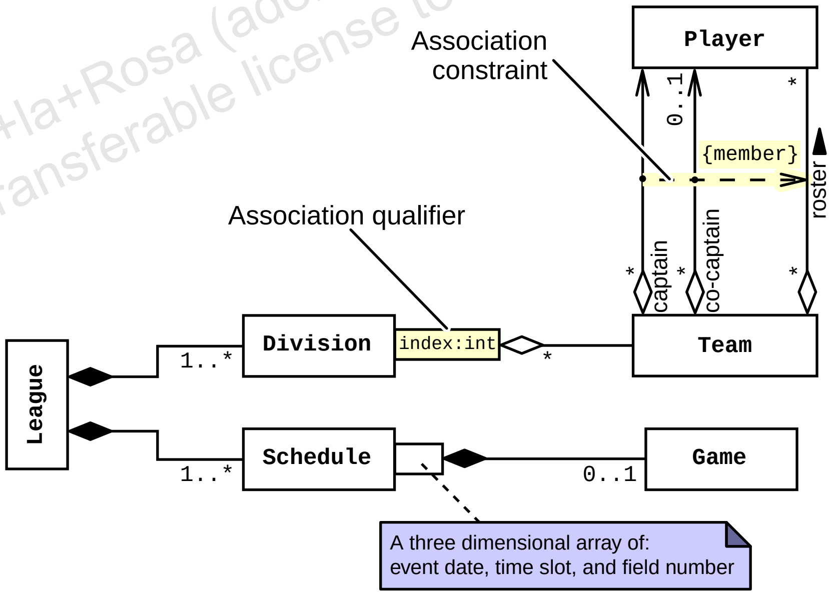


Figure E-16 Other Associations Elements

An association qualifier provides a modeling mechanism to state that an object at one end of the association can look up another object at the other end. For example, a particular game in a schedule is uniquely identified by an event date, such as Saturday August 12, 2000, a time-slot, such as 11:00 a.m. to 12:30 p.m., and a particular field number, such as field #2). One particular implementation might be a three dimension array (for example, `Game[][][]`), in which each qualifier element (date, time-slot, field#) is mapped to an integer index.

Object Diagrams

An Object diagram represents the static structure of a system at a particular instance in time. These diagrams are composed of object nodes, associations, and sometimes class nodes.

Figure E-17 shows a hierarchy of objects that represent a set of teams in a single division in a soccer sports league. This diagram shows one configuration of objects at a specific point of time in the system. Object nodes only show instance attributes because methods are elements of the class definition. Also, an Object diagram does need not to show every associated object, it just needs to be representative.

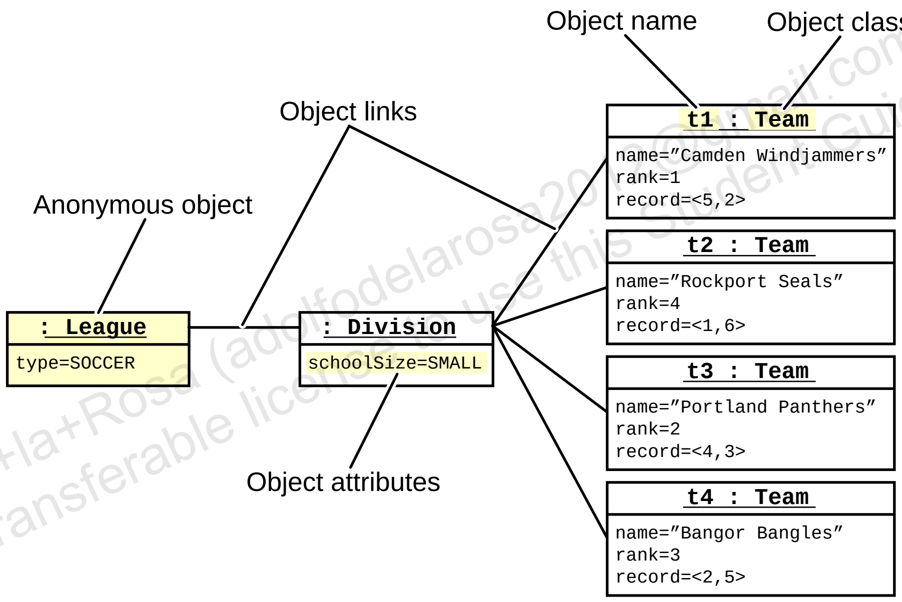


Figure E-17 An Example Object Diagram

Figure E-18 shows two objects, c1 and c2, with their instance data. They refer to the class node for Count, and the dependency arrow indicates that the object is an instance of the class Count. The objects do not include the counter attribute because it has class scope.

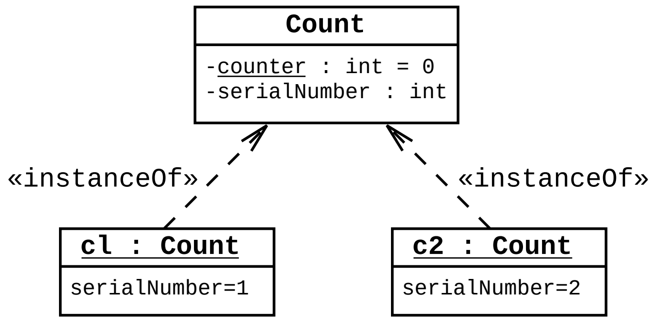


Figure E-18 An Example Object Diagram

Communication Diagrams

A Communication diagram represents a particular behavior shared by several objects. These diagrams are composed of objects, their links, and the message exchanges that accomplish the behavior.

Figure E-19 shows a Communication diagram in which an actor initiates a login sequence within a web application using a servlet.

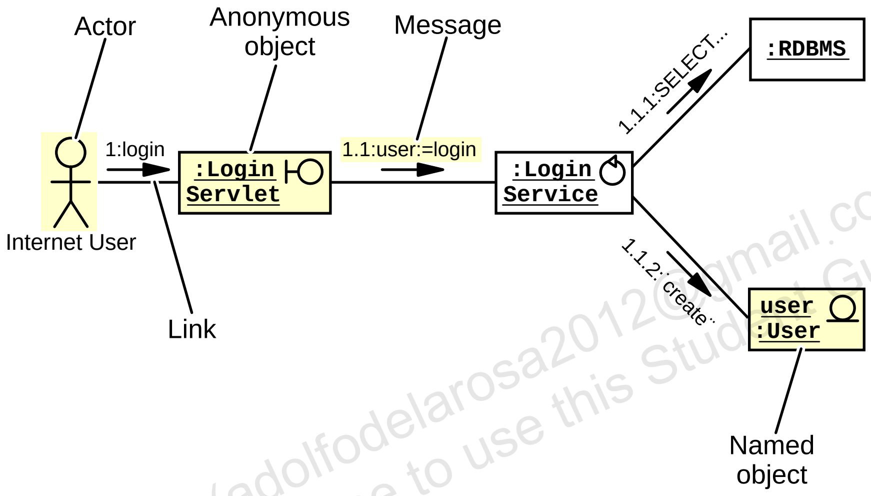


Figure E-19 An Example User-driven Communication Diagram

The servlet uses an object of the `LoginService` class to perform the lookup of the username, verify the password, and create the `User` object. The links between objects show the dependencies and collaborations between these objects. Messages between objects are shown by the messages on the links. Messages are indicated by an arrow in the direction of the message, and a text string declares the type of message. The text of this message string is unrestricted. Messages are also labeled with a sequence number so you can see the order of the message calls.

Figure E-20 shows a more elaborate Communication diagram. In this diagram, some client object initiates an action on a session bean. This session bean then performs two database modifications within a single transaction context.

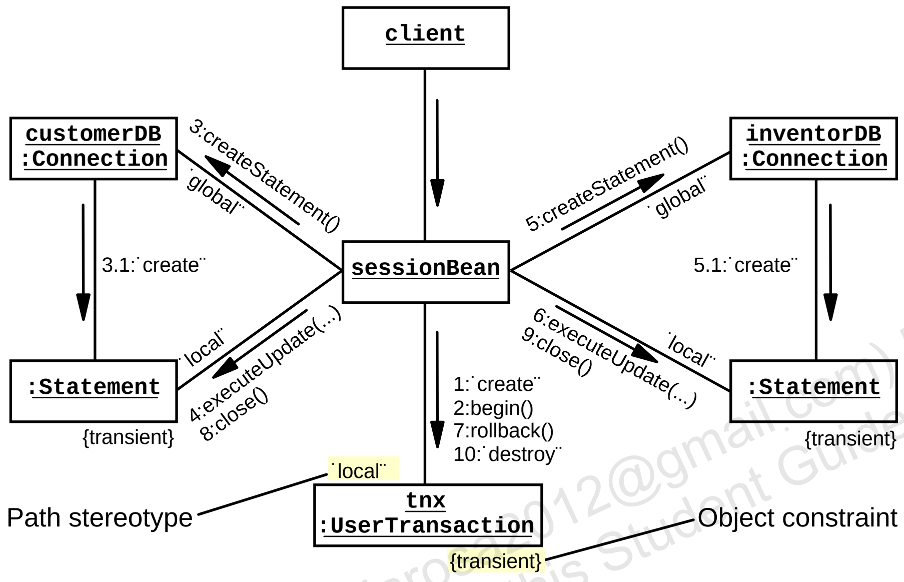


Figure E-20 Another Communication Diagram

You can label the links with a stereotype to indicate if the object is global or local to the call sequence. In this example, the connection objects are global, and the statement and transaction objects are local.

You can also label objects with a constraint to indicate if the object is transient.

Sequence Diagrams

A Sequence diagram represents a time sequence of messages exchanged between several objects to achieve a particular behavior. A Sequence diagram enables you to understand the flow of messages and events that occur in a certain process or collaboration. In fact, a Sequence diagram is just a time-ordered view of a Collaboration diagram (see page E-24).

Figure E-21 shows a Sequence diagram in which an actor initiates a login sequence within a web application which uses a servlet. This diagram is equivalent to the Collaboration diagram in Figure E-19 on page E-24. An important aspect of this type of diagram is that time is moving from top to bottom. The Sequence diagram shows the time-based interactions between a set of objects or roles. Roles can be named or anonymous and usually have a class associated with them. Roles can also be actors.

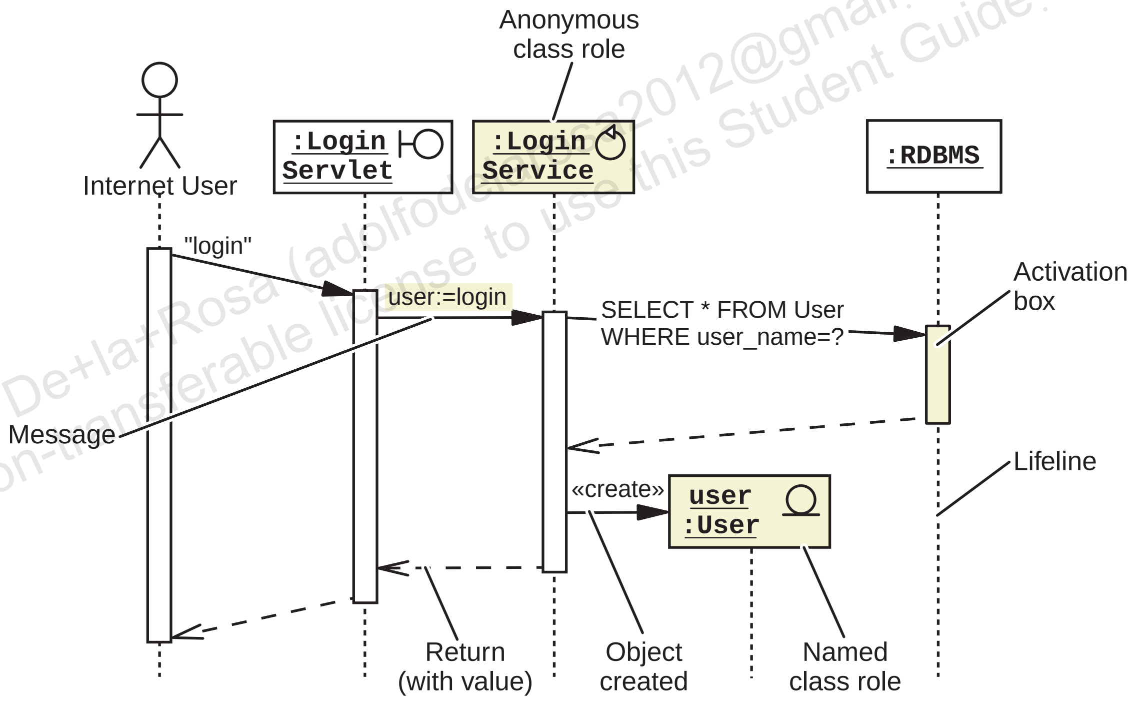


Figure E-21 An Example User-driven Sequence Diagram

Notice the message arrows between the servlet and the service object. The arrow is perfectly horizontal, which indicates that the message is probably implemented by the local method call. Notice that the message arrow between the actor and the servlet is angled, which indicates that the message is sent between components on different machines, such as an HTTP message from the user's web browser to the web container that handles the login servlet.

Figure E-22 shows a more elaborate Sequence diagram. This diagram is equivalent to the Communication diagram in Figure E-20 on page E-25.

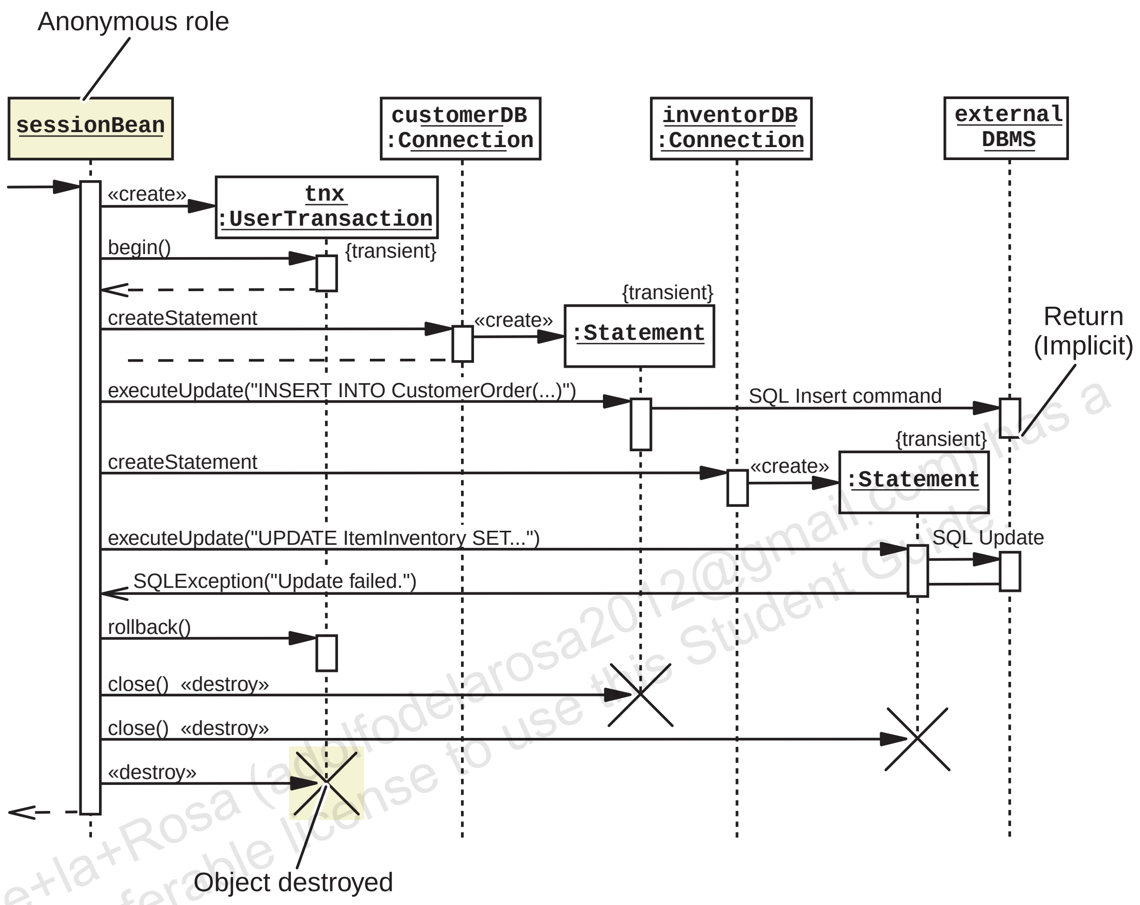


Figure E-22 Another Sequence Diagram

This example shows a few more details about Sequence diagrams. First, the return arrow is not always important. A return arrow is implicit at the end of the activation bar. Also, Sequence diagrams can show the creation and destruction of objects explicitly. Every role has a lifeline that extends from the base of the object node vertically. Roles at the top of the diagram existed before the entry message (into the left-most role). Roles that have a message arrow pointing to the head of the role node with the «create» message are created during the execution of the sequence. The destruction of an object is shown with a large cross that terminates the role's lifeline.

Note – Sequence diagrams can also show asynchronous messages. This type of message uses a solid line with stick arrow head. $\longrightarrow$

State Machine Diagrams

A State Machine diagram represents the states and responses of a class to external and internal triggers. You can also use a State Machine diagram to represent the life cycle of an object. The definition of an object state is dependent on the type of object and the level of depth you want to model.

Note – A State Machine diagram is recognized by several other names including Statechart diagram, State diagram, and State Transition diagram.

Figure E-23 shows an example State Machine diagram. Every State Machine diagram should have an initial state (the state of the object at its creation) and a final state. By definition, no state can transition into the initial state and the final state cannot transition to any other state.

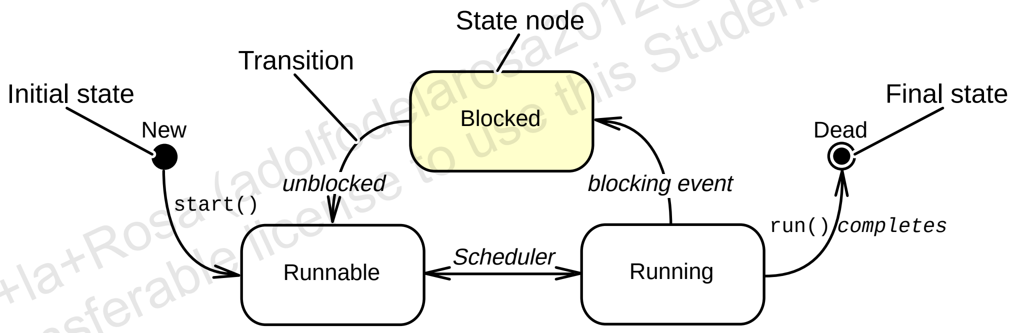


Figure E-23 An Example State Machine Diagram

There is no pre-defined way to implement a State Machine diagram. For complex behavior, you might consider using the State design pattern.

Transitions

A transition has five elements:

- Source state – The state affected by the transition
- Event trigger – The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
- Guard condition – A Boolean expression used to determine if the state transition should be made when the event trigger occurs
- Action – A computation or operation performed on the object that makes the state transition
- Target state – The state that is active after the completion of the transition

Figure E-24 illustrates a detailed transition.

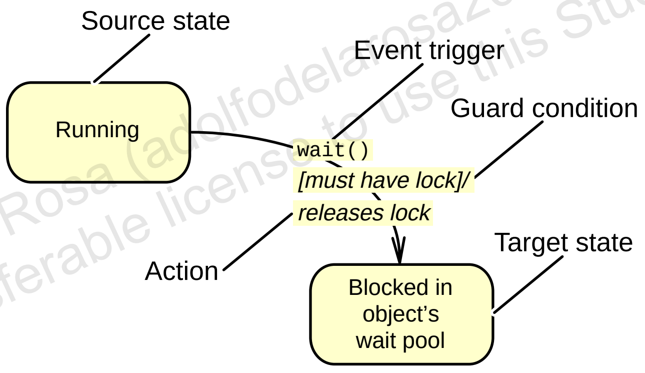


Figure E-24 An Example State Transition

Activity Diagrams

An Activity diagram represents the activities or actions of a process without regard to the objects that perform these activities.

Figure E-25 shows the elements of an Activity diagram. An Activity diagram is similar to a flowchart. There are activities and transitions between them. Every Activity diagram starts with a single start state and ends in a single stop state.

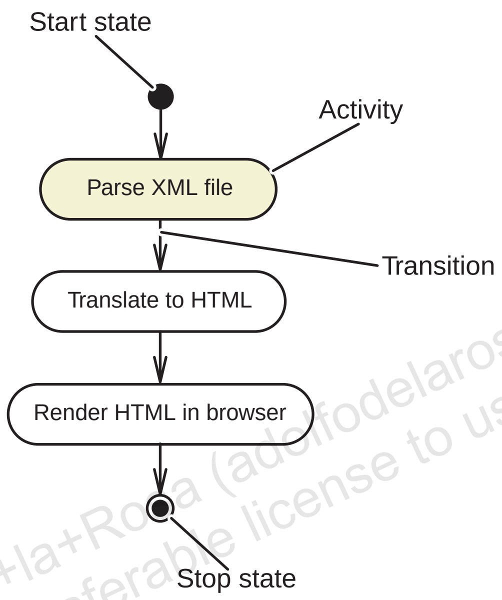


Figure E-25 Activities and Transitions

Figure E-26 demonstrates branching and looping in Activity diagrams. The diagram models the higher level activity of "Verify availability" of products in a purchase order. The top-level branch node forms a simple while loop. The guard on the transition below the branch is true if there are more products in the order to be processed. The "else" transition halts this activity.

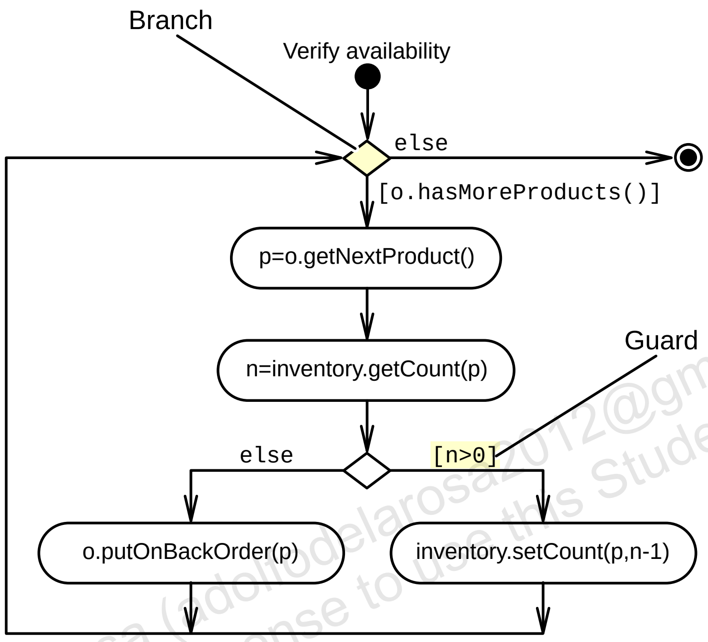


Figure E-26 Branching and Looping

Figure E-27 shows a richer Activity diagram.

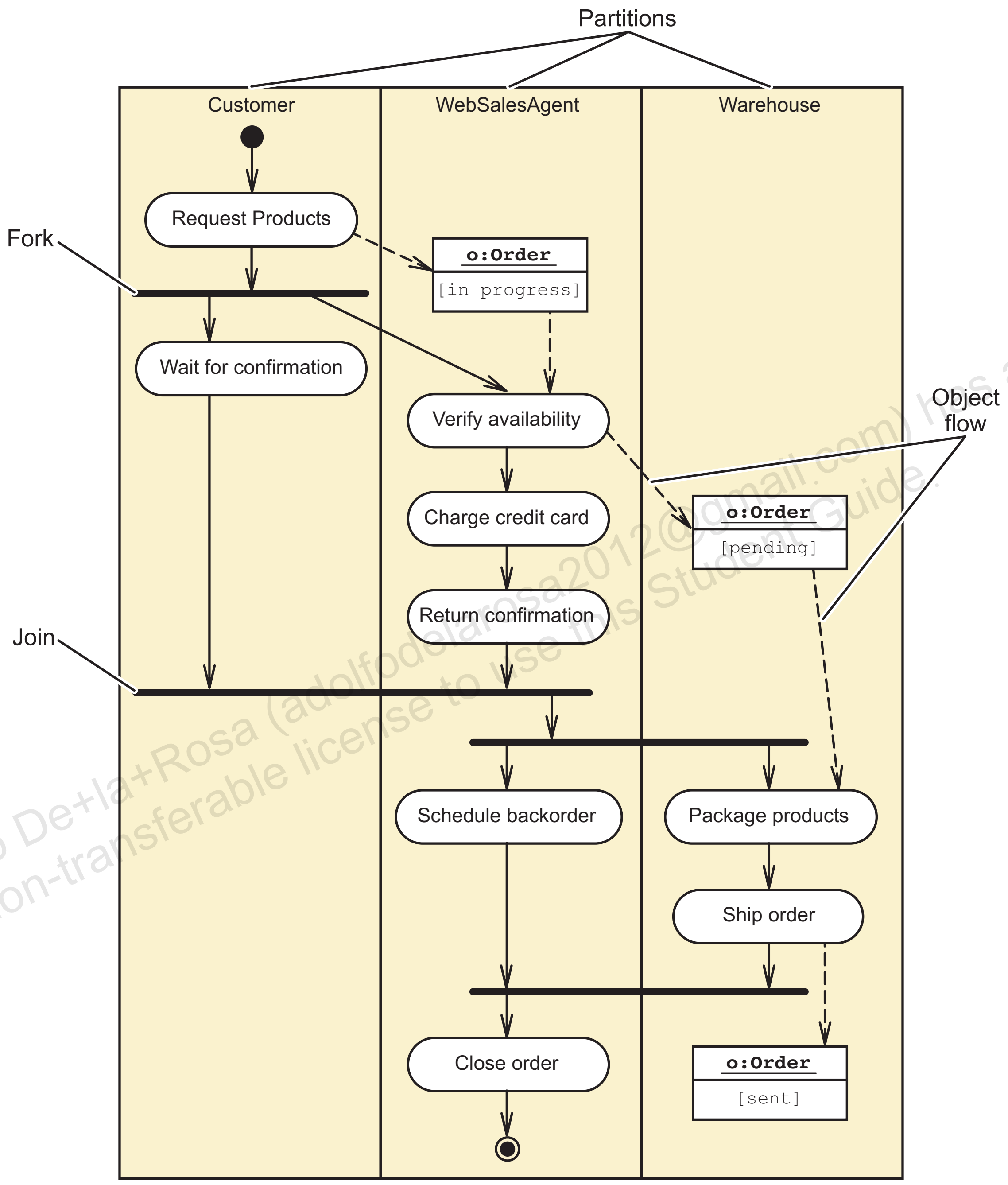


Figure E-27 An Example Activity Diagram

In this example, partitions (formerly called swim lanes) are used to isolate the actor of a given set of activities. These actors might include humans, systems, or organization entities. This diagram also demonstrates the ability to model concurrent activities. For example, the Customer initiates the purchase of one or more products on the company's web site. The Customer then waits as the WebSalesAgent software begins to process the purchase order. The fork bar splits a single transition into two or more transitions. The corresponding join bar must contain the same number of inbound transitions.

Figure E-28 shows a UML version 2 style of notation using object flow pins. A pin is represented as a small rectangle attached to an action (a non-divisible activity) with a label indicating the type of object that will flow though the pin. The pin label can also show the state of the object using the square brackets notation.

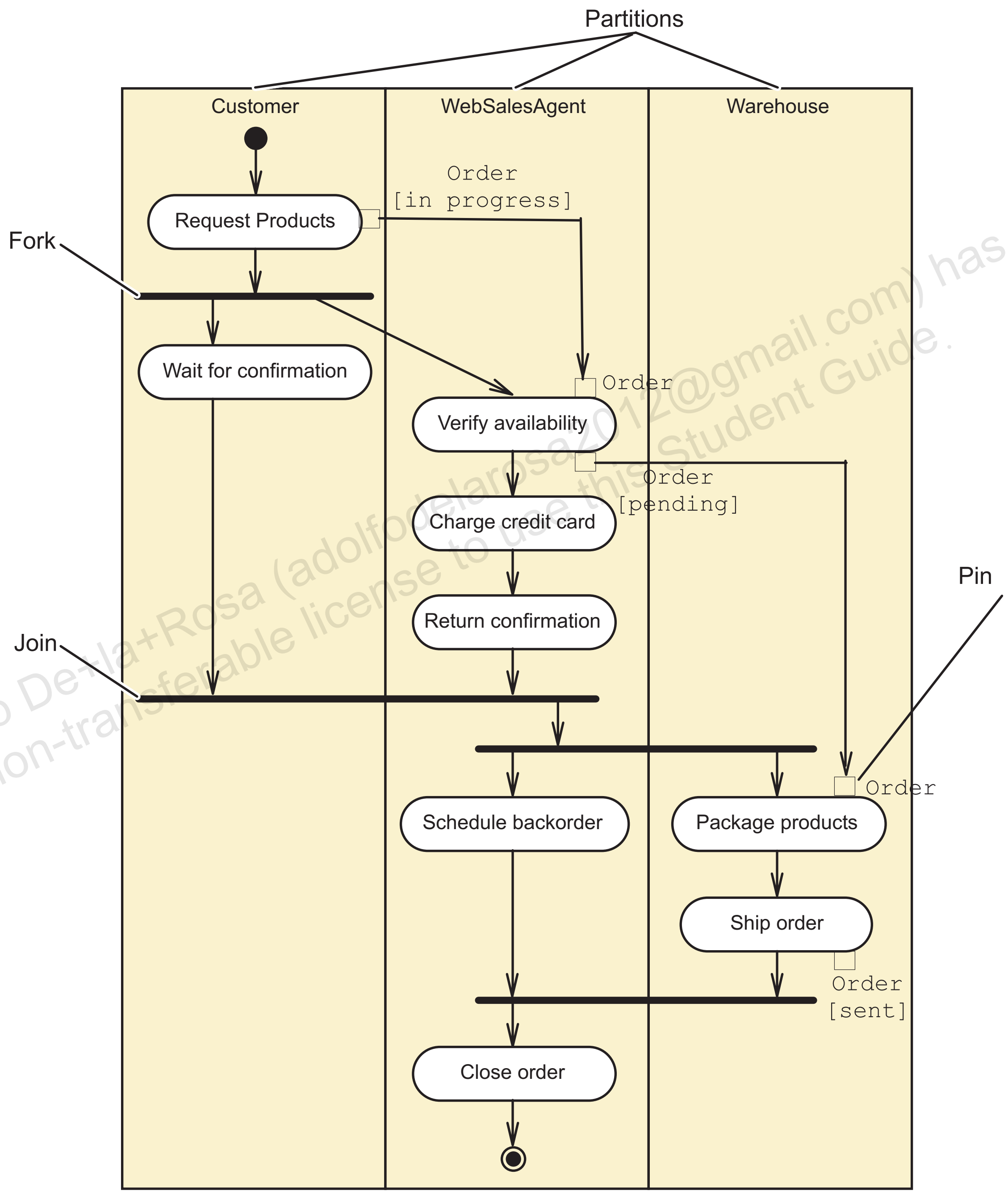


Figure E-28 An Example Activity Diagram Showing UML2 Object Flow Pins

Component Diagrams

A Component diagram represents the organization and dependencies among software implementation components.

Figure E-29 shows four types of icons that can represent software components. Example 1 is an icon that represents a generic component using the original UML notation. Example 2 is an icon that is used to represent a source file. Example 3 is an icon that represents a file that contains executable (or object) code. Example 4 is an icon that represents a generic component using UML version 2 notation.

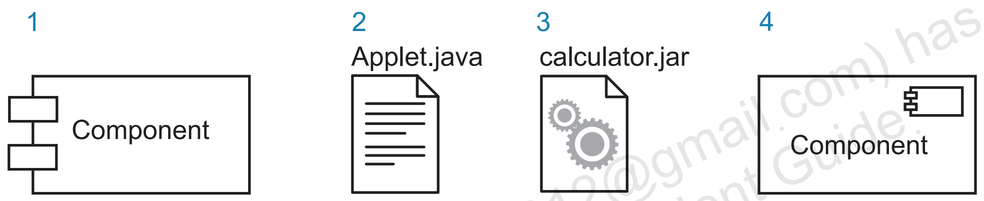


Figure E-29 Example Component Nodes

Figure E-30 shows the dependencies of packaging an HTML page that contains an applet. The HTML page depends on the JAR file, which is constructed from a set of class files. The Class files are compiled from the corresponding source files. You can use a tagged value to indicate the source control version numbers on the source files.

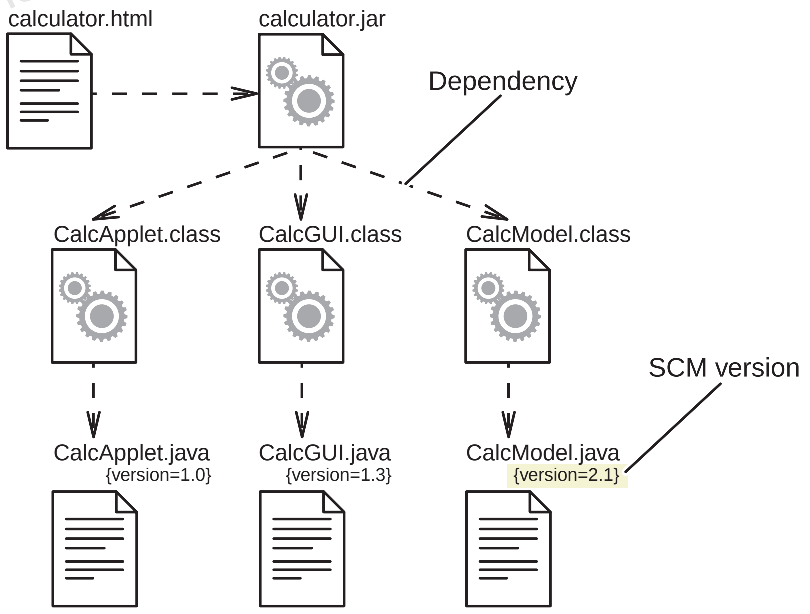


Figure E-30 An Example Component Diagram

Note – SCM is Source Control Management.

Figure E-31 shows another Component diagram. In this diagram, several components have an interface connector. The component attached to the connector implements the named interface. The component that has an arrow pointing to the connector depends on the fact that component realizes that interface.

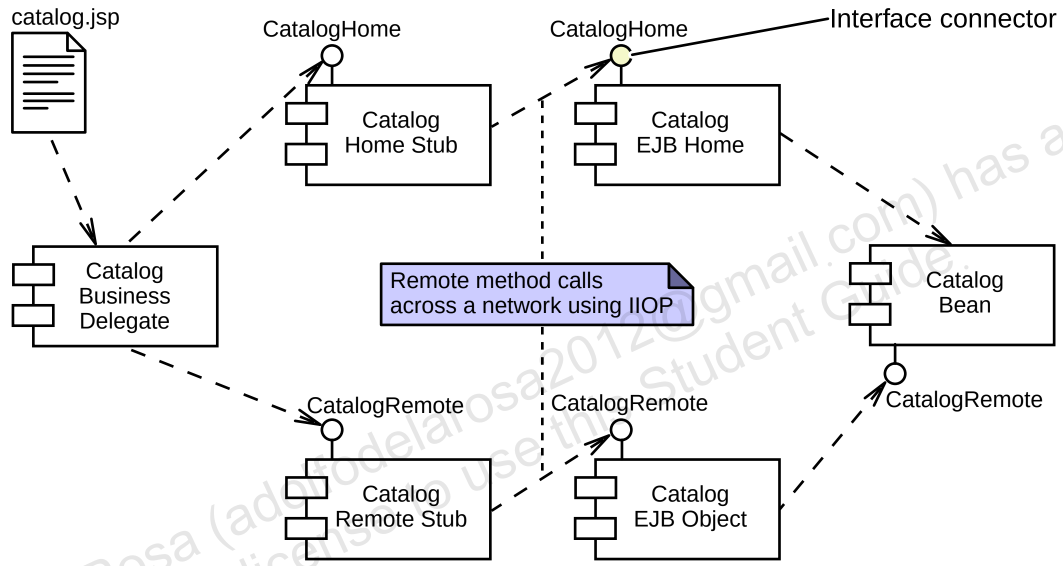


Figure E-31 A Component Diagram With Interfaces

In this J2EE technology example, the web tier includes a catalog JSP, which uses a catalog Business Delegate JavaBeans component to communicate to the EJB technology tier. Every enterprise bean must include two interfaces. The home interface enables the client to create new enterprise beans on the EJB server. The remote interface enables the client to call the business logic methods on the (remote) enterprise bean. The business delegate communicates with the catalog bean through local stub objects that implement the proper home and remote interfaces. These objects communicate over a network using the Internet Inter-ORB protocol with remote "skeletons." In EJB technology terms, these objects are called EJBHome and EJBObject). These objects communicate directly with the catalog bean that implements the true business logic.

Deployment Diagrams

A Deployment diagram represents the network of processing resource elements and the configuration of software components on each physical element.

A Deployment diagram is composed of hardware nodes, software components, software dependencies, and communication relationships. Figure E-32 shows an example in which the client machine communicates with a web server using HTTP over TCP/IP.

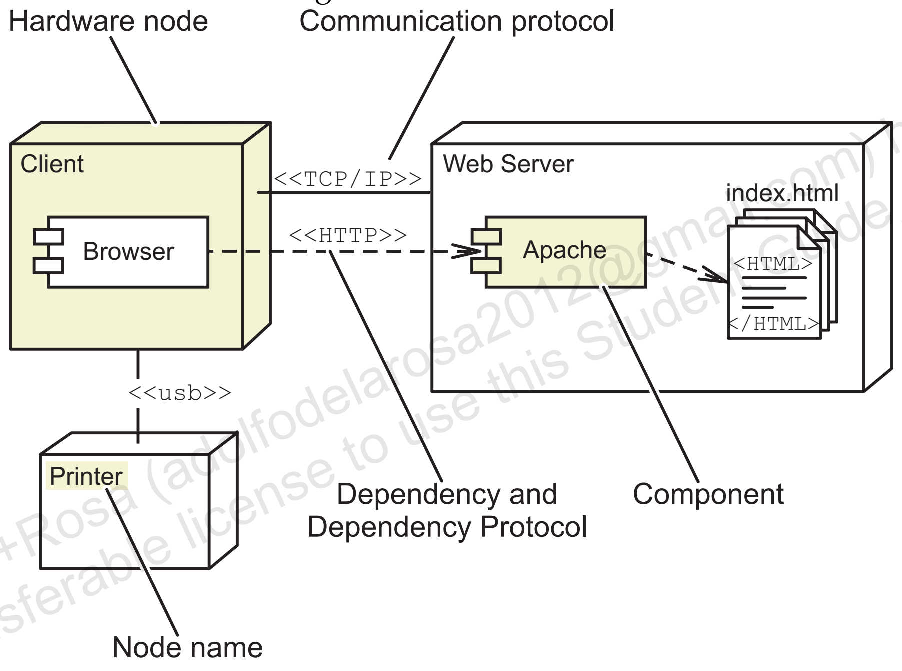


Figure E-32 An Example Deployment Diagram

The client is running a web browser which is communicating with an Apache web server. Therefore, the browser component depends upon the Apache component. Similarly, the Apache application depends upon the HTML files that it serves. The client machine is also connected to local printer using a parallel port.

You can use a Deployment diagram to show how the logical tiers of an application architecture are configured into a physical network.

Appendix F

Additional Resources

Additional Resources

The following references provide additional information on the topics described in this course:

- Alexander, Christopher. *A Pattern Language: Towns Buildings Construction*. Oxford University Press, Inc., 1977.
- Ambler, Scott. "Mapping Objects to Relational Databases," [`http://www.ambysoft.com/mappingObjects.pdf`]. accessed 2 October 2002.
- Arlow, Jim, and Ila Neustadt. *UML and the Unified Process*. Reading: Addison Wesley Longman, Inc., 2002.
- Arnold, Ken, James Gosling, and David Holmes. *The Java Programming Language (Third Edition)*. Boston: Addison Wesley Longman, Inc., 2000.
- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Upper Saddle River: Addison Wesley Longman, 1998.
- Beck, Kent. *eXtreme Programming eXplained*. Reading: Addison Wesley Longman, Inc., 2000.
- Beck, Kent, and Ward Cunningham. "A Laboratory For Teaching Object-Oriented Thinking." [`http://c2.com/doc/oopsla89/paper.html`] accessed 5 October 2002.
- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Booch, Grady. *Object-Oriented Analysis and Design with Applications (2nd ed).* The Benjamin/Cummins Publishing Company, Inc., Redwood City, 1994.

- Booch, Grady. *Object Solutions (Managing the Object-Oriented Project).* Reading: Addison Wesley Longman, Inc., 1994.
- Brooks, Frederick. *The Mythical Man-month (anniversary edition)*. Reading: Addison Wesley Longman, Inc., 1995.
- 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.
- Chillarege, Ram. "Software Testing Best Practices." [`http://www.chillarege.com/authwork/TestingBestPractice.pdf`]. accessed 18 October 2002.
- Cockburn, Alistair. *Agile Software Development*. Reading: Addison Wesley Longman, Inc., 2001.
- Folwer, Martin. *Refactoring (Improving the Design of Existing Code)*. Reading: Addison Wesley Longman, Inc., 2000.
- Folwer, Martin, with Kendall Scott. *UML Distilled (2nd ed)*. Reading: Addison Wesley Longman, Inc., 2000.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Gulutzan, Peter, Trudy Pelzer. *SQL-99 Complete, Really*. Lawrence: R&D Books, 1999.
- Grosso, William. *Java RMI*. Sebastopol: O'Reilly & Associates, Inc. 2002.
- Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Reading: Addison-Wesley. 1999.
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Harlow: Addison Wesley Longman, Inc., 1993.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, 2000.
- 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.
- Norman, Donald A. *The Design of Everyday Things*. New York: Currency/Doubleday, 1988.

- The Object Management Group. “Unified Modeling Language (UML), Version 2.2” [`http://www.omg.org/technology/documents/formal/uml.htm`].
- Pressman, Roger. *Software Engineering A Practitioner’s Approach, Fifth edition*. McGraw-Hill, 2001.
- Rosenberg, Doug, and Kendall Scott. *Use Case Driven Object Modeling with UML (A Practical Approach)*. Reading: Addison Wesley Longman, Inc., 1999.
- Rosenberg, Doug, and Kendall Scott. *Applying Use Case Driven Object Modeling with UML (An Annotated e-Commerce Example)*. Reading: Addison Wesley Longman, Inc., 2001.
- 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.
- Sun Microsystems, Inc. *Java Look and Feel Design Guidelines*. Reading: Addison Wesley, 1999.
- Sun Microsystems, Inc. *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*. [`http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html`], accessed 6 October 2002.
- Sun Microsystems, Inc. *Glossary of Java technology-related terms*. [`http://java.sun.com/docs/glossary.html`], accessed 4 December 2002.
- 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.
- Vlissides, John, James Coplien, and Norman Kerth. *Pattern Language of Program Design (vol. 2)*. Reading: Addison Wesley Longman, Inc., 1996.
- 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)*. Upper Saddle River: Prentice Hall, 2005.
- Fowler, Martin. *Analysis Patterns*. Addison Wesley Longman, Inc., 1997.

- 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.

Glossary/Acronyms

This glossary includes both terms and acronyms. The entry for an acronym shows the expanded phrase; the definition is found under the entry for the phrase. If a definition came from an external source, a citation is provided for the source after the definition.

An entry might contain multiple definitions. If the definitions are merely variations on the same concept, then each definition will reside on its own text line. If the definitions are different, then each definition is numbered and a category is presented in parenthesis. For example:

constraint

1. (SD) Any condition placed on the development of a project, such as what programming language or technologies to use.
2. (UML) A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. (UML v1.4 page B-6)

The set of categories used in this glossary are shown in Table 1

Table 1 Definition Categories

Category Abbreviation	Category Definition
Arch	architecture
GUI	graphical user interfaces
Java	Java technology terms
SD	software development
SW	software (general)
UML	Unified Modeling Language

A

abstract class

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

A class that cannot be directly instantiated. (UML v1.4 page B-2)

(see *abstract method*)
(antonym: *concrete class*)

abstract coupling

Abstract coupling exists when one class depends on either an abstract class or an interface.

abstract method

A method that has no implementation. (Sun Glossary)

abstraction

The essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer. (UML v1.4 page B-2)

Abstract Window Toolkit

A collection of graphical user interface (GUI) components that were implemented using native-platform versions of the components. (Sun Glossary)

ACM

Association for Computing Machinery

active class

A class whose instances are active objects. (UML v1.4 page B-2)

(see *active object*)

active object

An object that owns a thread and can initiate control activity. An instance of active class. (UML v1.4 page B-2)

activities

The specific actions of one or more workers that produce an artifact. This is the lowest level of organization of the software development process. This describes the who and what of the details of the SD process.

Activity diagram

A UML diagram depicting a flow of activities that might be performed by either a system or an actor.

actors

Users (human or machine) of software.

A coherent set of roles that users of use cases play when interacting with these use cases. (UML v1.4 page B-3)

acyclic dependency principle

The dependencies between packages must form no cycles. (Knoernschild page 27)

ADP

(see *acyclic dependency principle*)

aggregation

A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. (UML v1.4 page B-3)

Agile methodologies

"a useful compromise between no process and too much process" (Fowler, `http://www.martinfowler.com/articles/newMethodology.html`) (see also `http://www.agilealliance.org/`)

Analysis model

The model that "refines the use cases in more detail and makes an initial allocation of behavior of the system to a set of objects that provides the behavior." (Jacobson USPD page 9)

analysis paralysis

The condition in which the development team spends an inordinate amount of time in the Requirements Gathering and Analysis workflows.

API

(see *application programming interface*)

application

A single, deployable component that offers a coherent set of use cases to an end user.

Application layer

In a layered architecture, the Application layer contains the components that are bought or built to support the FRs of the system.

application programming interface

The specification of how a programmer writing an application accesses the behavior and state of classes and objects. (Sun Glossary)

architecturally significant use cases

The use cases of a system that are considered risky (especially, technical risks).

Architecture baseline

The code base that implements the architecturally significant use cases, supports the non-functional requirements, and mitigates the project risks. This is the end result of the Elaboration phase.

Architecture-centric

A software development approach in which the "architecture is defined and validated before the development teams begin the bulk of implementing the system design." (Sun SunTone page 21)

A software development approach in which the "system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development." (Jacobson USDP page 443)

Architecture model

The model that includes the details of the infrastructure components that are required to support the constraints and NFRs of the proposed system.

Architecture patterns

Software patterns applied at a high-level. These patterns usually solve problems that affect the NFRs of the system.

Architecture template

An abstract model of the detailed Deployment diagram in which the Analysis components are abstracted to a single element.

Architecture workflow

The workflow that models the highest level structure of the system. The architecture model must satisfy the NFRs.

artifact

A physical piece of information that is used or produced by a software development process. (UML v1.4 page B-3)

A tangible product of the development process. (Booch Object Solutions page 303)

assembler paradigm

The programming paradigm in which the software is written in assembler (machine language) for a specific hardware and operating system platform.

association
: The semantic relationship between two or more classifiers that specifies connections among their instances. (UML v1.4 page B-3)

association class
: A model element that has both association and class properties. (UML v1.4 page B-4)

attribute
: A feature within a classifier (such as a class) that describes a range of values that instances of the classifier may hold. (UML v1.4 page B-4)

availability
: The system quality that measures the amount of time the system can process a request.

AWT
: (see *Abstract Window Toolkit*)

B

B2B
: business-to-business

best practice
: Recommended techniques that have stood the test of time.

Business Analyst role
: The person who gathers requirements from the client stakeholders and analyzes the functional requirements by modeling the enduring business themes of the system.

business entity
: (see *Domain entity*)

business logic
: The code that supports the workflow of a use case and other functional requirements.

Business Owner role
: The lead stakeholder or client of the project. The business owner is responsible for making final decisions about the behavior of the system.

business process
: Any coherent collection of activities performed by an person or a system that produces a result of value.

Business tier
The tier whose services "*execute business logic and manage transactions*." (Sun SunTone AM page 15)

C

CACM
Communications of the ACM

CCP
(see *common closure principle*)

CICS
Customer Information Control System, an IBM product

CIO
Chief Information Officer

class
A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. (UML v1.4 page B-5)

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

Class diagram
A UML diagram depicting a collection of software classes and their inter-relationships.

client
1. (SD) The person or group that is responsible for defining the requirements of the propose system.

2. (SW) The object that uses the services of another object.

client/server
A system that is composed of two physical tiers in which the client tier requests services of a server tier.

Client tier
The tier that contains any "device or system that manages display and local interaction processing." (Sun SunTone AM page 15)

Communication diagram
A UML diagram representing a collection of objects that work together to support some system behavior.

common closure principle

Classes that change together, belong together (in a package). (Knoernschild page 174)

Common Object Request Broker Architecture

A language independent, distributed object model specified by the Object Management Group. (Sun Glossary)

component

A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. (UML v1.4 page B-5)

component, Boundary

A component representing a user interface element. This is a primary component in an Analysis model.

component, Controller

A component representing the control aspects of a user interface, such as accepting user input and actions.

Component diagram

A diagram that shows the organizations and dependencies among components. (UML v1.4 page B-6)

component, Entity

A component representing a Domain entity. This is a primary component in an Analysis model.

component, GUI

A component representing an element in a GUI screen, such as a text field or a button.

component, Service

A component which encapsulates a coherent set of business operations, usually representing a use case workflow. This is a primary component in an Analysis model. In the UP method, this component type is called "Control."

component, View

A component representing the visual aspects of a user interface.

composite reuse principle

Favor polymorphic composition of objects over inheritance. (Knoernschild page 17)

Composite Structure diagram

A UML diagram representing the internal structure of a classifier, usually in form of parts, and can include the interaction ports and interfaces (provided or required).

composition

A form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts. (UML v1.4 page B-6)

(see *aggregation*)

concrete class

A class in which all methods have been defined and in which instantiation is permitted.

(see *class*)
(antonym: *abstract class*)

constraint

1. (SD) Any condition placed on the development of a project, such as what programming language or technologies to use.

2. (UML) A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. (UML v1.4 page B-6)

Construction phase

A phase of the Unified Software Development Process focusing on building the software.

constructor

A method-like member of a class that initializes the attributes of an object being instantiated.

container

1. (Arch) A component that exists to contain other components. (UML v1.4 page B-6)

   For example, Tomcat is an application-level component that acts as a web container which manages the life cycle of servlet and JSP components within a web application.

2. (SW) An object that holds a collection of other objects, such as an array, list, set, or map.

3. (GUI) A GUI component that groups other GUI components. A GUI container might be a complete window or a panel within a window.

COO
Chief Operational Officer

CORBA
(see *Common Object Request Broker Architecture*)

CTO
Chief Technology Officer

coupling
The degree with which two classes depend upon each other.

CPR
(see *composite reuse principle*)

CPU
central processing unit

CRC analysis
CRC stands for Class-Responsibility-Collaboration. CRC analysis is a technique for identifying key abstractions in the problem domain.

CRUD
This acronym represents the four fundamental database operations: Create, Retrieve, Update, Delete.

D

DAO
(see *Data Access Object*)

Data Access Object
A software pattern that separates the handling of object persistence from the business logic classes.

database
(see *database management system*)

database management system
A system (or embedded subsystem) that provides data persistence facilities.

data type
(see *type*)

DB
database

(see *database management system*)

DBMS

(see *database management system*)

(also see *RDBMS* and *OODBMS*)

DDL

Data Definition Language

dependency

A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element). (UML v1.4 page B-7)

dependency inversion principle

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

Deployment diagram

A UML diagram depicting a collection of components distributed across one or more hardware nodes.

Deployment Specialist role

The person who deploys the implementation onto the production platform.

Deployment workflow

The workflow that puts the implementation into production.

derived attribute

An attribute that can be computed from one or more attributes.

Design pattern

Software patterns applied at a medium-level. These patterns usually solve problems that affect the FRs of the system.

Design workflow

The workflow that generates the Solution model.

Developmental qualities

Developmental qualities are the systemic qualities that are reflected in the immediate development of the system. This category includes such qualities as realizability and planability.

development plan

The schedule of activities for a software project, often based on a prioritized list of use cases.

development team

The group of people that develop the software solution.

DIP
: (see *dependency inversion principle*)

discipline
: A new term originating from the OMG as a replacement for the term *workflow*.

distributed system
: Any system that requires two or more components to communicate across physical hardware nodes.

DLL
: dynamically linked library

domain entity
: A synonym of *key abstraction*.

Domain model
: A Class diagram of the key abstractions of the problem domain space.

DTD
: (XML) document type definition

dynamic binding
: The ability of the runtime environment to determine which method to call.

E

EBT
: (see *enduring business themes*)

EIS
: Enterprise Information System

EJB
: (see *Enterprise JavaBean*)

Elaboration phase
: A phase of the Unified Software Development Process focusing on creating an architecture baseline upon which the rest of the software system will be constructed.

encapsulation
: (see *information hiding*)

enduring business themes
The functions of a company that, year after year (decade after decade), do not change.
(see also *key abstraction*)

Enterprise JavaBean
A business-logic component that complies with the EJB specification.

entity
A synonym of *key abstraction*.

entity-relation
The dominate modeling technique for visualizing database schemas.

ER
(see *entity-relation*)

ER diagram
A diagram that represents the entity relationships of the database schema.

ER diagram, logical
An ER diagram in which many details have been left out.

ER diagram, physical
An ER diagram in which the details have been specified. These details include field datatypes, key constraints, indexes, and so on.

event
(GUI) A object that represents a user action, such as moving the mouse, pressing a button, typing on the keyboard, and so on.

event listener
(GUI) A object that responds to user events.

evolutionary prototype
This is the code that is used to prototype and test the Architecture model. This prototype becomes the Architecture baseline after all risks have been mitigated.

Evolutionary qualities
Evolutionary qualities are the systemic qualities that are reflected in the total cost of ownership of the system. This category includes such qualities as scalability, reusability, extensibility, and so on.

extensibility
The system quality that measures the effort saved when adding new functionality.

eXtreme Programming

This methodology enables customers to rank-order features and to change their minds without recrimination from the tech staff. It emphasizes quick-release cycles of code, a focus on keeping the system as simple as possible, and constant testing.

(see also *Agile methodologies*)

F

flexibility

The system quality that measures the effort saved when implementing some change.

forward engineering

The process of creating code (or at least a code skeleton) from a set of models.

FR

(see *functional requirement*)

framework

A collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms that clients can use or adapt. (Booch page 514)

functional paradigm

The programming paradigm in which the software is written as a collection of interacting functions with no side-effects.

functional requirement

The set formal and informal descriptions of the behavior of the system from the user's perspective. This maps closely to the problem model.

G

GANTT chart

A Gantt chart provides a graphical illustration of a schedule that helps to plan, coordinate, and track specific tasks in a project.

generalization

A taxonomic relationship between a more general element and a more specific element. (UML v1.4 page B-10)

In the UML, this relationship is denoted by a solid line with a solid triangular arrowhead.

(antonym: *generalization*)

GoF

Gang of Four. The four authors of the *Design Patterns* book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

graphical user interface

A user interface that uses windows and GUI components to provide a rich user experience.

guard (condition)

1. (SW) A Boolean expression that permits or prevents flow of control.

2. (UML) *A condition that must be satisfied in order to enable an associated transition to fire.* (UML v1.4 page B-10)

GUI

(see *graphical user interface*)

guillemet

This is a quotation symbol in some European languages such as French. The open guillemet is « and the close guillemet is ». These symbols are used in UML to quote the names of stereotypes, for example «`refines`».

H

Hardware layer

In a layered architecture, the Hardware layer describes the physical characteristics of the machines that support the upper layers.

hierarchy

An ordered classification of classes based on inheritance.

(also know as: taxonomy)

hierarchy, whole-part

(see *whole-part hierarchy*)

HP

Hardware Platform layer

HR

human resources

HRS

Hotel Reservation System

HTML

(see *HyperText Markup Language*)

HTML form

A collection of GUI components generated in a web page which enables the user to enter data and submit requests to the web server.

HTTP

(see *HyperText Transfer Protocol*)

HTTPS

A version of HTTP that uses a secure socket layer (SSL) to provide confidentiality and data integrity for message sent across the Internet.

HVAC

heating, ventilation, and air conditioning

Hypertext Markup Language

This is a file format, based on SGML, for hypertext documents on the Internet. (Sun Glossary)

Hypertext Transfer Protocol

The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts. (Sun Glossary)

I

ID

identifier

idiom

Software patterns applied at a low-level. These patterns usually solve problems within the scope of a certain computer language, platform, or technology.

IIOP

Internet Inter-ORB Protocol

(see also *CORBA*)

Implementation workflow

The workflow that realizes the solution model. The implementation *model* (which is really just the source code) should have a one-to-one mapping to the solution model. That is to say that the implementation can be coded from the solution model.

Inception phase

A phase of the Unified Software Development Process focusing on understanding the business case for the proposed system.

information hiding

The ability to prevent certain aspects of a class from being accessible to its clients. (Meyer page 1197) Refers to hiding implementation details behind a public interface (one or more methods).

infrastructure

The internal structure of a software system.

infrastructure component

A component that supports the infrastructure of a software system. These components usually do not directly support functional requirements.

inheritance

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

Integration tier

The tier whose services "abstract and process access to external resources." (Sun SunTone AM page 15)

Interaction Overview diagram

A UML diagram representing 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.

interface

A named set of operations that characterize the behavior of an element. (UML v1.4 page B-11)

interface, Java technology

An interface is like a class but has only declarations of it methods. (Arnold, Gosling, and Holmes page 26)

A Java technology interface is roughly equivalent to a C++ class in which all of the methods are empty virtuals.

Internet

An enormous network consisting of literally millions of hosts from many organizations and countries around the world. It is physically put together from many smaller networks and data travels by a common set of protocols. (Sun Glossary)

Internet protocol

The basic protocol of the Internet. It enables the unreliable delivery of individual packets from one host to another. (Sun Glossary)

intranet

A company-wide network.

IP

(see *Internet protocol*)

IT

Information Technology

iterative development

In the context of the software life cycle, a process that involves managing a stream of executable releases. (Jacobson USDP page 446)

Iterative development focuses on growing the system in small, incremental, and planned steps. (Knoernschild page 77)

J

J2EE

(see *Java 2 Platform, Enterprise Edition*)

J2ME

(see *Java 2 Platform, Mobile Edition*)

J2SE

(see *Java 2 Platform, Standard Edition*)

JAR

An archive file that holds a structured collection of Java technology classes and other files. The structure is based on the package hierarchy of the classes.

Java 2 Platform, Enterprise Edition

This is a specification for a platform that supports enterprise application developments. This is a superset of J2SE.

An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multi-tiered, Web-based applications. (Sun Glossary)

Java 2 Platform, Mobile Edition

This is a specification for a platform that supports the development of software for small devices, such as palm computers and cell phones. This is a subset of J2SE.

Java 2 Platform, Standard Edition

The core Java technology platform. (Sun Glossary)

JavaBean

A portable, platform-independent reusable component model. (Sun Glossary)

JavaServer Pages™

An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser. (Sun Glossary)

Java Virtual Machine

A software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor (whether in a computer or in another electronic device). (Sun Glossary)

JDBC

JDBC stands for Java DataBAse Connectivity. An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC provides a call-level API for SQL-based database access. (Sun Glossary)

job role

The responsibility of a type of worker.

JRMP

Java Remote Method Protocol

(see also *RMI*)

JSP

(see *JavaServer Pages*)

JVM

(see *Java Virtual Machine*)

K

key abstraction

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

key, compound

A database key that is composed of more than one field in the table.

key, foreign

A set of fields in one table that uniquely identifies a single row in another table. A foreign key in the former table is usually the primary key in the latter table.

key, primary

A set of fields in a table that uniquely identifies a single row in the table.

L

layer

The hardware and software stack that hosts services within a give tier. (Sun SuntTone AM page 10)
(also see *tier*)

LDAP

Lightweight Directory Access Protocol

legacy system

An older, potentially moldy system that must be preserved for any number of economic or social reasons, yet must also coexist with newly developed elements. (Booch Object Solutions page 305)

link

A connection between two objects; an instance of an association. (Larman page 617)

Lower Platform layer

In a layered architecture, the Lower Platform layer describes the operating system upon which the upper layers depend.

LP

(see *Lower Platform layer*)

M

maintainability

The system quality that measures the effort saved during revision and correction of design flaws.

maintenance

A revision of software embodying the whole development process.

manageability

The system quality that measures the decrease in effort of performing minor administrative tasks.

Manifest qualities

Manifest qualities are the systemic qualities that are reflected in the execution of the system as experienced by a single user. This category includes such qualities as performance, usability, availability, and so on.

mental model

A model of some system that exists in a person's mind.

message

A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation. (UML v1.4 page B-11)

method

A function or procedure that is applied to an object. A method implements a message.

methodology

"A body of methods, rules, and postulates employed by a discipline." Software methodology is the highest level project organization. A methodology puts a repeatable structure into the software development process.

method signature

The specification of the interface to a method. A signature usually includes the name of the method, the parameters, and the return type.

model

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)

Model 2 architecture

A Presentation tier structure that uses a variation on the MVC pattern, in which servlets act as a Controller and JSP pages act as Views.

mouse

1. A computer peripheral device that moves a cursor on the monitor.

2. A small fury rodent.

multiplicity

A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. (UML v1.4 page B-13)

MVC

Model-View-Controller, an architecture pattern

N

navigation

(UML) The ability to traverse an object association in a specific direction. The direction of navigation is specified by the arrowhead(s) on the association in a Class diagram.

NFR

(see *non-functional requirements*)

non-functional requirements

The set formal and informal descriptions of the systemic qualities that the system must satisfy, as well as the technological constraints that the implementation must follow.

n-tier system

A system that is composed of multiple physical tiers.

O

object

object = state + behavior

An object is a runtime instance of a class that contains attributes and operations.

An entity with a well-defined boundary and identity that encapsulates state and behavior. (UML v1.4 page B-13)

object association

(see *association*)

object database

(see *OODBMS*)

Object diagram

A UML diagram depicting a runtime *snapshot* of software objects and their inter-relationships.

Object Management Group

A not-for-profit organization that promotes the research and development of object technologies. The OMG maintains the specifications for CORBA and UML. (`http://www.omg.org/`)

Object Modeling Technique

An object modeling language and method developed in the late 1980's by James Rumbaugh.

object-oriented
A philosophy of software design in which objects (and not procedures) are the central organizational structure.

object-oriented paradigm
The programming paradigm in which the software is written as a collection of interacting objects passing messages to each other.

OCP
(see *open closed principle*)

OH
Overhead slide book

OMG
(see *Object Management Group*)

OMT
(see *Object Modeling Technique*)

OO
(see *object-oriented*)

OOAD
Object-Oriented Analysis and Design

OODBMS
A database management system that stores entities as objects. (also see *DBMS* and *RDBMS*)

OOSD
Object-Oriented Software Development

OOSE
Object-Oriented Software Engineering

open closed principle
Classes should be open for extension but closed for modification. (Knoernschild page 8)

operating system
The system and set of utilities that provide applications with support to execute code, interact with peripherals, and communicate with other external systems.

operation
Some behavior of an object. This usually co-responds to a method.

Operational qualities

Operational qualities are the systemic qualities that are reflected in the execution of the system. This category includes such qualities as throughput, security, serviceability, and so on.

OS

(see *operating system*)

P

PAC

Presentation-Abstraction-Control, an architecture pattern

package

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

2. (Java) A hierarchical naming structure that organizes a group of *types*.

Package diagram

A UML diagram depicting a collection of other modeling elements and diagrams.

pattern

(see *software pattern*)

PDA

personal digital assistant

performance

The system quality that measures how quickly the system fulfills a user request.

persistence

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

phases

The highest level of organization of the time-dimension in SD process. UP defines four phases: Inception, Elaboration, Construction, and Transition. Each phase includes one or more iterations.

planability

The system quality that measures the confidence that a system can be planned with appropriate cost estimations.

polymorphism

A concept in type theory, according to which a name (such as a variable declaration) might denote objects of many different classes that are related by some common superclass [type]. (Booch OOA&D with Apps page 517)

portability

The system quality that measures the effort saved when migrating to a different infrastructure (such as a new lower platform).

Presentation tier

The tier whose services "aggregate and personalize content and services into channel-specific user interfaces." (Sun SunTone AM page 15)

procedural paradigm

The programming paradigm in which the software is written as a hierarchy of procedures.

Profile diagram

A UML diagram that might define additional diagram types or extend existing diagrams with additional notations.

project glossary

A glossary that records the terminology of the problem domain. The project glossary is usually included in the SRS document.

Project Manager role

The person who manages aspects of a software development project, such as budget, resources, and schedule.

prototype

A small, coherent collection of code that supports some small-scale goal, such as proving the utility of a certain technology.

PST

Pacific standard time

Q

QA

quality assurance

qualified association

An attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

quality of service
: A measure of the qualitative characteristics (such as performance, reliability, and scalability) of a system.

R

RAM
: random access memory

Rational Unified Process
: A software development process based on the Unified Process, but supported by a tool set from Rational Software, Inc.

RDBMS
: A database management system that stores entities as rows in tables. (also see *DBMS* and *OODBMS*)

realizability
: The system quality that measures the probability (or confidence) that the proposed system can be built.

realization
: A relationship between an interface and the class that implements that interface. In the UML, this relationship is denoted by a dashed line with a solid triangular arrowhead (a "dashed generalization" symbol).

refactoring
: A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. (Fowler Refactoring page 53)

relational database
: (see *RDBMS*)

relational schema
: The definition of tables, fields, indexes, and so on, for a database.

reliability
: The system quality that measures the frequency of correctness in an operation.

requirement
: A condition or capability to which a system must conform. (Jacobson USDP page 448)

Requirements Analysis workflow

The workflow that models the problem domain. OR: The workflow that generates the problem model. The problem model is based solely on the functional requirements

Requirements Gathering workflow

The workflow that generates the documentation to describe, in both formal and informal descriptions, the business problem.

Requirements model

The model that defines how the software system is intended to behave. This model includes both FRs and NFRs as well as other constraints on the system.

Resource tier

The tier which includes "legacy systems, databases, external data feeds, specialized hardware devices such as telco switches or factory automation, and so on." (Sun SunTone AM page 16)

reusability

The system quality that measures the effort gained by leveraging existing components for new purposes.

reverse engineering

The process of creating a model from a collection of existing code.

risk

A project variable that endangers or eliminates success for a project. (Jacobson USDP page 448)

RMI

Remote Method Invocation

RMI-IIOP

Remote Method Invocation over IIOP

Robustness analysis

Robustness analysis is a technique for identifying the components of the system that supports one or more use cases.

ROI

return on investment

role

The named specific behavior of an entity participating in a particular context. A role may be static (for example, an association end) or dynamic (for example, a collaboration role). (UML v1.4 page B-16)

row

A single entry in a relational database table.

rule-based paradigm

The programming paradigm in which the software is written as a collection of interacting rules and fact-base. Such systems tend to be goal-driven in which rules are used to deduce new information from existing facts.

RUP

(see *Rational Unified Process*)

S

SAP

(see *stable abstractions principle*)

scalability

The system quality that measures the ratio of load growth required to the cost to implement that capacity.

SD

(see *software development*)

SDK

(see *software development kit*)

SDP

(see *stable dependencies principle*)

security

The system quality that prevents undesired use (misuse or abuse) of the system.

Separation of Concerns

An architectural principle in which different components are created to support different purposes within the software. For example, MVC is a pattern that supports the separation of the business logic (Model) components from the user interface (View and Controller) components.

Sequence diagram

A UML diagram depicting a time-oriented perspective of an object collaboration.

serviceability

The system quality that measures the effort required to update or repair the system.

servlet

A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web clients using a request-response paradigm. (Sun Glossary)

Simple Object Access Protocol

SOAP is a uses a combination of XML-based data structuring and HTTP to define a standardized method for invoking methods in objects distributed in diverse operating environments across the internet. (Sun Glossary)

SOAP

(see Simple Object Access Protocol)

software

The set of instructions that directs the actions of a computer system.

Software Architect role

The person who defines the architecture of the system, leads the development of the architectural baseline during the Inception and Elaboration phases, analyzes the non-functional requirements, and identifies project risks and creates a risk mitigation plan.

Software Designer role

The person who creates the solution model of the system based on the functional requirements (use cases) within the framework of the architecture.

software development

The set of activities that support the creation of software.

software development kit

A set of tools (and possibly one or more frameworks) that enable an engineer to construct software systems.

software pattern

A description of communicating objects and classes that are customized to solve a general design problem in a particular context. (Gamma, Helm, Johnson, Vissides page 3)

A repeatable solution to a recurring problem in a given context.

Software Programmer role

The person who implements the software solution.

Solution model

The model that describes the software components that satisfy both the functional and non-functional requirements of a use case.

specialization
(antonym: *generalization*)

SRS
(see: *System Requirements Specification*)

stable abstractions principle
Stable packages should be abstract packages. (Knoernschild page 31)

stable dependencies principle
Depend on the direction of stability. (Knoernschild page 29)

State Machine diagram
A UML diagram depicting a set of states that an object might experience and the triggers that transition the object from one state to another state.

stakeholder
Any party (person or group) that has an interest in the project.

stereotype
A new type of modeling element that extends the semantics of the metamodel. (UML v1.4 page B-18)

Structured Query Language
The standardized relational database language for defining database objects and manipulating data. (Sun Glossary)

SQL
(see *Structured Query Language*)

state
1. (UML) A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. (UML v1.4 page B-17)

2. (SW) The configuration of attribute values in an object.

state transition
The process of changing state within an object.

subclass
A class that is derived from a particular class, perhaps with one or more classes in between. (Sun Glossary)

subsystem
A grouping of model elements that represents a behavioral unit in a physical system. (UML v1.4 page B-19)

A collection of modules, some of which are visible to other subsystems and other of which are hidden. (Booch OOAD page 519)

subtype

If type X extends or implements type Y, then X is a subtype of Y. (Sun Glossary)

SunTone Architecture Methodology

A software development process based on "a step-by-step process for creating dot-com (n-tier and web-centric) architectures." (Sun SunTone page 4)

superclass

A class from which a particular class is derived, perhaps with one or more classes in between. (Sun Glossary)

supertype

The supertypes of a type are all the interfaces and classes that are extended or implemented by that type. (Sun Glossary)

system

A collection of interacting software and hardware components usually at the application-level.

system boundary

In a UML Use Case diagram, the box that separates the use cases for a system from the outside of the system (the actors).

System Requirements Specification

A document that defines all requirements (both FRs and NFRs) for a system.

systemic qualities

The term used by the SunTone architecture methodology for non-functional requirements.
(see also *quality of service*)

systemic-quality-driven

An software development approach that "places a critical emphasis on identifying, ranking, and quantifying the systemic qualities as requirements." (Sun SunTone page 23)

T

table

A coherent collection of rows (all with the same fields) in a relational database.

tag

A syntactic structure in an SGML (XML or HTML) file that identifies a specific piece of information.

TCP

(see *transmission control protocol*)

testability

The system quality that measures the effort required to identify and isolate a fault or error in the system.

Test Engineer role

The person who tests the implementation to verify that the system meets the requirements (both functional and non-functional).

Testing workflow

The workflow in which you test the implementation against the expectations as defined by the requirements.

thread

The basic unit of program execution. A process can have several threads running concurrently, each performing a different job, such as waiting for events or performing a time-consuming job that the program does not need to complete before going on. (Sun Glossary)

threadsafe

The condition of a method (or set of methods in a component) in which there

throughput

The system quality that measures the amount of work done by the system, measured in operations per unit time.

tier

A logical or physical organization of components into an ordered chain of service providers and consumers. (Sun SunTone AM page 10)

Timing diagram

A UML diagram representing changes in state (state lifeline view) or value (value lifeline view). It can also show time and duration constraints and interactions between timed events.

transaction

An atomic unit of work that modifies data. A transaction encloses one or more program statements, all of which either complete or roll back. Transactions enable multiple users to access the same data concurrently. (Sun Glossary)

(abbreviated: txn)

Transition phase

A phase of the Unified Software Development Process focusing on readying the software for production.

transmission control protocol

This is an Internet protocol that provides for the reliable delivery of streams of data from one host to another. (Sun Glossary)

type

A stereotyped class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. (UML v1.4 page B-20)

A class or interface. (Sun Glossary)

type, primitive

A data type that is not part of the object types of a computer language. These usually include support for integers, real numbers (called floating-point numbers), characters, Boolean values, and so on.

U

UC

(see *use case*)

UI

(see *user interface*)

UML

(see *Unified Modeling Language*)

UML tool

An tool that supports the creation and maintenance of UML diagrams for a project.

Unified Modeling Language

A standard modeling language for software – a language for visualizing, specifying, constructing, and documenting artifacts of a software-intensive system. (Jacobson USDP page 449)

Unified Software Development Process

A software development process based on the UML that is iterative, architecture-centric, use-case-driven, and risk-driven. (Jacobson USDP page 449)

UP

(see *Unified Software Development Process*)

Upper Platform layer
: In a layered architecture, the Upper Platform layer contains the systems that implement the Virtual Platform specifications. The systems in this layer are usually containers for components in the Application layer.

US
: United States

usability
: The system quality that measures the ease by which a user can accomplish some goal.

USDP
: (see Unified Software Development Process)

use case
: The activities performed by software for actors to support a single system function. Often called a "user story."

Use Case diagram
: An UML diagram depicting the set of high-level behaviors the system must perform for a given actor.

Use-Case-driven
: In the context of the software life cycle, meaning that use cases are used as a primary artifact for establishing the desired behavior of the system and for communicating this among the stakeholders of the system. (Jacobson USDP page 450)

An software development approach in which "every attempt is made to prioritize design and development around the realization of complete, end-to-end use cases." (Sun SunTone page 21)

use case form
: A text form that records the analysis of a use case.

use case scenario
: A text story of a specific instance of a use case.

user interface
: The boundary components that interact with the user.

V

view
: A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective. (UML v1.4 page B-21)

Virtual Platform layer

In a layered architecture, the Virtual Platform layer contains the specifications and APIs upon which the Application layer components depend.

visibility

An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing name space. (UML v1.4 page B-21)

Vision document

A document that records the main ideas of a proposed software system that usually includes the business case for building such a system.

VP

virtual platform

VPN

virtual private network

W

W3C

World Wide Web Consortium

WAE

Web Application Extension

Waterfall

A software development process based on the a single iteration through the SD workflows.

web application

A coherent collection of dynamically-generated web pages and forms that supports one or more use cases of a system.

web container

A container that provides the network services over which requests and responses are sent, decodes requests, and formats responses. All servlet (web) containers must support HTTP as a protocol for requests and responses, but may also support additional request-response protocols such as HTTPS. (Sun Glossary)

WebMVC

Web model-view-controller

web server

A host machine that supports one or more Web sites or web applications.

web services

A distributed remote procedure communication mechanism.

Web site

A coherent collection of web pages at a single web server.

Web tier

(synonym: *Presentation tier*)

WebUI

(see *web user interface*)

web user interface

A UI that is presented in a Web browser screen.

whole-part hierarchy

An ordered classification of objects based on the breakdown of parts and subparts.

window

(GUI) A rectangular portion of the computer monitor that contains user interface components.

worker

A person that performs an activity.

workflow

1. The middle level of organization of activities within an iteration. The classical SD workflows are: requirements gathering, requirements analysis, architecture, design, implementation, test, and deployment.

2. A business process.

X

XML

eXtensible Markup Language

XP

(see *eXtreme Programming*)