



**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and**

**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and**

**Iterative Development, Third Edition**

**Iterative Development, Third Edition**

By Craig Larman

By Craig Larman

**Publisher**: Addison Wesley Professional

**Publisher**: Addison Wesley Professional

**Pub Date**: October 20, 2004

**Pub Date**: October 20, 2004

**ISBN**: 0-13-148906-2

**ISBN**: 0-13-148906-2

• Table of

• Table of Contents

Contents • Index

• Index

**Pages**: 736

**Pages**: 736

*Applying UML and Patterns* is the world's #1 business and college

*Applying UML and Patterns* is the world's #1 business and college introduction to "thinking in objects"and using that insight in real-world

introduction to "thinking in objects"and using that insight in real-world object-oriented analysis and design. Building on two widely acclaimed

object-oriented analysis and design. Building on two widely acclaimed previous editions, Craig Larman has updated this book to fully reflect the

previous editions, Craig Larman has updated this book to fully reflect the new UML 2 standard, to help you master the art of object design, and to

new UML 2 standard, to help you master the art of object design, and to promote high-impact, iterative, and skillful agile modeling practices.

promote high-impact, iterative, and skillful agile modeling practices. Developers and students will learn object-oriented analysis and design

Developers and students will learn object-oriented analysis and design (OOA/D) through three iterations of two cohesive, start-to-finish case

(OOA/D) through three iterations of two cohesive, start-to-finish case studies. These case studies incrementally introduce key skills, essential OO

studies. These case studies incrementally introduce key skills, essential OO principles and patterns, UML notation, and best practices. You won't just

principles and patterns, UML notation, and best practices. You won't just learn UML diagramsyou'll learn how to *apply* UML in the context of OO

learn UML diagramsyou'll learn how to *apply* UML in the context of OO software development.

software development.

Drawing on his unsurpassed experience as a mentor and consultant,

Drawing on his unsurpassed experience as a mentor and consultant, Larman helps you understand evolutionary requirements and use cases,

Larman helps you understand evolutionary requirements and use cases, domain object modeling, responsibility-driven design, essential OO design,

domain object modeling, responsibility-driven design, essential OO design, layered architectures, "Gang of Four" design patterns, GRASP, iterative

layered architectures, "Gang of Four" design patterns, GRASP, iterative methods, an agile approach to the Unified Process (UP), and much more.

methods, an agile approach to the Unified Process (UP), and much more. This edition's extensive improvements include

This edition's extensive improvements include

A stronger focus on helping you master OOA/D through case studies

A stronger focus on helping you master OOA/D through case studies that demonstrate key OO principles and patterns, while also applying

that demonstrate key OO principles and patterns, while also applying the UML

the UML

New coverage of UML 2, Agile Modeling, Test-Driven Development,

New coverage of UML 2, Agile Modeling, Test-Driven Development, and refactoring

and refactoring

Many new tips on combining iterative and evolutionary development

Many new tips on combining iterative and evolutionary development with OOA/D

with OOA/D

Updates for easier study, including new learning aids and graphics Updates for easier study, including new learning aids and graphics New college educator teaching resources

New college educator teaching resources

Guidance on applying the UP in a light, agile spirit, complementary

Guidance on applying the UP in a light, agile spirit, complementary with other iterative methods such as XP and Scrum

with other iterative methods such as XP and Scrum

Techniques for applying the UML to documenting architectures Techniques for applying the UML to documenting architectures A new chapter on evolutionary requirements, and much moreA new chapter on evolutionary requirements, and much more

***Applying UML and Patterns, Third Edition,*** is a lucid and practical

introduction to thinking and designing with objectsand creating systems

that are well crafted, robust, and maintainable.







**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and**

**Iterative Development, Third Edition**

By Craig Larman

**Publisher**: Addison Wesley Professional

**Pub Date**: October 20, 2004

**ISBN**: 0-13-148906-2

• Table of

Contents

• Index

Copyright

**Pages**: 736

Praise for Applying UML and Patterns

Contents by Major Topics

Foreword

Preface

Educator and Web Resources

Intended Audiencean Introduction!

Prerequisites

Java Examples, But …

Book Organization

About the Author

Contact

Enhancements to the Previous Edition

Acknowledgments

Typographical Conventions

Production Notes

Part 1. Introduction

Chapter 1. Object-Oriented Analysis and Design

Section 1.1. What Will You Learn? Is it Useful?

Section 1.2. The Most Important Learning Goal?

Section 1.3. What is Analysis and Design?

Section 1.4. What is Object-Oriented Analysis and Design? Section 1.5. A Short Example

Section 1.6. What is the UML?

Section 1.7. Visual Modeling is a Good Thing

Section 1.8. History

Section 1.9. Recommended Resources

Chapter 2. Iterative, Evolutionary, and Agile

Introduction

Section 2.1. What is the UP? Are Other Methods Complementary? Section 2.2. What is Iterative and Evolutionary Development? Section 2.3. What About the Waterfall Lifecycle?

Section 2.4. How to do Iterative and Evolutionary Analysis and Design? Section 2.5. What is Risk-Driven and Client-Driven Iterative Planning? Section 2.6. What are Agile Methods and Attitudes? Section 2.7. What is Agile Modeling?

Section 2.8. What is an Agile UP?

Section 2.9. Are There Other Critical UP Practices?

Section 2.10. What are the UP Phases?

Section 2.11. What are the UP Disciplines?

Section 2.12. How to Customize the Process? The UP Development Case Section 2.13. You Know You Didn't Understand Iterative Development or the UP When... Section 2.14. History

Section 2.15. Recommended Resources

Chapter 3. Case Studies

Introduction

Section 3.1. What is and isn't Covered in the Case Studies?

Section 3.2. Case Study Strategy: Iterative Development + Iterative Learning Section 3.3. Case One: The NextGen POS System

Section 3.4. Case Two: The Monopoly Game System

Part 2. Inception

Chapter 4. Inception is Not the Requirements Phase

Introduction

Section 4.1. What is Inception?

Section 4.2. How Long is Inception?

Section 4.3. What Artifacts May Start in Inception?

Section 4.4. You Know You Didn't Understand Inception When... Section 4.5. How Much UML During Inception?

Chapter 5. Evolutionary Requirements

Introduction

Section 5.1. Definition: Requirements

Section 5.2. Evolutionary vs. Waterfall Requirements

Section 5.3. What are Skillful Means to Find Requirements?

Section 5.4. What are the Types and Categories of Requirements? Section 5.5. How are Requirements Organized in UP Artifacts? Section 5.6. Does the Book Contain Examples of These Artifacts? Section 5.7. Recommended Resources

Chapter 6. Use Cases

Introduction

Section 6.1. Example

Section 6.2. Definition: What are Actors, Scenarios, and Use Cases? Section 6.3. Use Cases and the Use-Case Model

Section 6.4. Motivation: Why Use Cases?

Section 6.5. Definition: Are Use Cases Functional Requirements? Section 6.6. Definition: What are Three Kinds of Actors?

Section 6.7. Notation: What are Three Common Use Case Formats? Section 6.8. Example: Process Sale, Fully Dressed Style

Section 6.9. What do the Sections Mean?

Section 6.10. Notation: Are There Other Formats? A Two-Column Variation Section 6.11. Guideline: Write in an Essential UI-Free Style

Section 6.12. Guideline: Write Terse Use Cases

Section 6.13. Guideline: Write Black-Box Use Cases

Section 6.14. Guideline: Take an Actor and Actor-Goal Perspective Section 6.15. Guideline: How to Find Use Cases

Section 6.16. Guideline: What Tests Can Help Find Useful Use Cases? Section 6.17. Applying UML: Use Case Diagrams

Section 6.18. Applying UML: Activity Diagrams

Section 6.19. Motivation: Other Benefits of Use Cases? Requirements in Context Section 6.20. Example: Monopoly Game

Section 6.21. Process: How to Work With Use Cases in Iterative Methods? Section 6.22. History

Section 6.23. Recommended Resources

Chapter 7. Other Requirements

Introduction

Other Requirement Artifacts

Section 7.1. How Complete are these Examples?

Section 7.2. Guideline: Should We Analyze These Thoroughly During Inception? Section 7.3. Guideline: Should These Artifacts be at the Project Website? Section 7.4. NextGen Example: (Partial) Supplementary Specification Section 7.5. Commentary: Supplementary Specification

Section 7.6. NextGen Example: (Partial) Vision

Section 7.7. Commentary: Vision

Section 7.8. NextGen Example: A (Partial) Glossary

Section 7.9. Commentary: Glossary (Data Dictionary)

Section 7.10. NextGen Example: Business Rules (Domain Rules) Section 7.11. Commentary: Domain Rules

Section 7.12. Process: Evolutionary Requirements in Iterative Methods Section 7.13. Recommended Resources

Part 3. Elaboration Iteration 1 Basics

Chapter 8. Iteration 1Basics

Introduction

Section 8.1. Iteration 1 Requirements and Emphasis: Core OOA/D Skills Section 8.2. Process: Inception and Elaboration

Section 8.3. Process: Planning the Next Iteration

Chapter 9. Domain Models

Introduction

Section 9.1. Example

Section 9.2. What is a Domain Model?

Section 9.3. Motivation: Why Create a Domain Model?

Section 9.4. Guideline: How to Create a Domain Model?

Section 9.5. Guideline: How to Find Conceptual Classes? Section 9.6. Example: Find and Draw Conceptual Classes Section 9.7. Guideline: Agile ModelingSketching a Class Diagram Section 9.8. Guideline: Agile ModelingMaintain the Model in a Tool? Section 9.9. Guideline: Report ObjectsInclude 'Receipt' in the Model? Section 9.10. Guideline: Think Like a Mapmaker; Use Domain Terms Section 9.11. Guideline: How to Model the Unreal World? Section 9.12. Guideline: A Common Mistake with Attributes vs. Classes Section 9.13. Guideline: When to Model with 'Description' Classes? Section 9.14. Associations

Section 9.15. Example: Associations in the Domain Models Section 9.16. Attributes

Section 9.17. Example: Attributes in the Domain Models

Section 9.18. Conclusion: Is the Domain Model Correct?

Section 9.19. Process: Iterative and Evolutionary Domain Modeling Section 9.20. Recommended Resources

Chapter 10. System Sequence Diagrams

Introduction

Section 10.1. Example: NextGen SSD

Section 10.2. What are System Sequence Diagrams?

Section 10.3. Motivation: Why Draw an SSD?

Section 10.4. Applying UML: Sequence Diagrams

Section 10.5. What is the Relationship Between SSDs and Use Cases? Section 10.6. How to Name System Events and Operations?

Section 10.7. How to Model SSDs Involving Other External Systems? Section 10.8. What SSD Information to Place in the Glossary? Section 10.9. Example: Monopoly SSD

Section 10.10. Process: Iterative and Evolutionary SSDs

Section 10.11. History and Recommended Resources

Chapter 11. Operation Contracts

Introduction

Section 11.1. Example

Section 11.2. Definition: What are the Sections of a Contract? Section 11.3. Definition: What is a System Operation?

Section 11.4. Definition: Postconditions

Section 11.5. Example: enterItem Postconditions

Section 11.6. Guideline: Should We Update the Domain Model? Section 11.7. Guideline: When Are Contracts Useful?

Section 11.8. Guideline: How to Create and Write Contracts

Section 11.9. Example: NextGen POS Contracts

Section 11.10. Example: Monopoly Contracts

Section 11.11. Applying UML: Operations, Contracts, and the OCL Section 11.12. Process: Operation Contracts Within the UP

Section 11.13. History

Section 11.14. Recommended Resources

Chapter 12. Requirements to DesignIteratively

Introduction

Section 12.1. Iteratively Do the Right Thing, Do the Thing Right Section 12.2. Provoking Early Change

Section 12.3. Didn't All That Analysis and Modeling Take Weeks To Do? Chapter 13. Logical Architecture and UML Package Diagrams

Introduction

Section 13.1. Example

Section 13.2. What is the Logical Architecture? And Layers?

Section 13.3. What Layers are the Focus in the Case Studies? Section 13.4. What is Software Architecture?

Section 13.5. Applying UML: Package Diagrams

Section 13.6. Guideline: Design with Layers

Section 13.7. Guideline: The Model-View Separation Principle Section 13.8. What's the Connection Between SSDs, System Operations, and Layers? Section 13.9. Example: NextGen Logical Architecture and Package Diagram Section 13.10. Example: Monopoly Logical Architecture?

Section 13.11. Recommended Resources

Chapter 14. On to Object Design

Introduction

Section 14.1. Agile Modeling and Lightweight UML Drawing

Section 14.2. UML CASE Tools

Section 14.3. How Much Time Spent Drawing UML Before Coding? Section 14.4. Designing Objects: What are Static and Dynamic Modeling? Section 14.5. The Importance of Object Design Skill over UML Notation Skill Section 14.6. Other Object Design Techniques: CRC Cards

Chapter 15. UML Interaction Diagrams

Introduction

Section 15.1. Sequence and Communication Diagrams

Section 15.2. Novice UML Modelers Don't Pay Enough Attention to Interaction Diagrams! Section 15.3. Common UML Interaction Diagram Notation

Section 15.4. Basic Sequence Diagram Notation

Section 15.5. Basic Communication Diagram Notation

Chapter 16. UML Class Diagrams

Introduction

Section 16.1. Applying UML: Common Class Diagram Notation

Section 16.2. Definition: Design Class Diagram

Section 16.3. Definition: Classifier

Section 16.4. Ways to Show UML Attributes: Attribute Text and Association Lines Section 16.5. Note Symbols: Notes, Comments, Constraints, and Method Bodies Section 16.6. Operations and Methods

Section 16.7. Keywords

Section 16.8. Stereotypes, Profiles, and Tags

Section 16.9. UML Properties and Property Strings

Section 16.10. Generalization, Abstract Classes, Abstract Operations Section 16.11. Dependency

Section 16.12. Interfaces

Section 16.13. Composition Over Aggregation

Section 16.14. Constraints

Section 16.15. Qualified Association

Section 16.16. Association Class

Section 16.17. Singleton Classes

Section 16.18. Template Classes and Interfaces

Section 16.19. User-Defined Compartments

Section 16.20. Active Class

Section 16.21. What's the Relationship Between Interaction and Class Diagrams? Chapter 17. GRASP: Designing Objects with Responsibilities

Section 17.1. UML versus Design Principles

Section 17.2. Object Design: Example Inputs, Activities, and Outputs Section 17.3. Responsibilities and Responsibility-Driven Design

Section 17.4. GRASP: A Methodical Approach to Basic OO Design Section 17.5. What's the Connection Between Responsibilities, GRASP, and UML Diagrams? Section 17.6. What are Patterns?

Section 17.7. Where are We Now?

Section 17.8. A Short Example of Object Design with GRASP

Section 17.9. Applying GRASP to Object Design

Section 17.10. Creator

Section 17.11. Information Expert (or Expert)

Section 17.12. Low Coupling

Section 17.13. Controller

Section 17.14. High Cohesion

Section 17.15. Recommended Resources

Chapter 18. Object Design Examples with GRASP

Introduction

Section 18.1. What is a Use Case Realization?

Section 18.2. Artifact Comments

Section 18.3. What's Next?

Section 18.4. Use Case Realizations for the NextGen Iteration Section 18.5. Use Case Realizations for the Monopoly Iteration Section 18.6. Process: Iterative and Evolutionary Object Design Section 18.7. Summary

Chapter 19. Designing for Visibility

Introduction

Section 19.1. Visibility Between Objects

Section 19.2. What is Visibility?

Chapter 20. Mapping Designs to Code

Introduction

Section 20.1. Programming and Iterative, Evolutionary Development Section 20.2. Mapping Designs to Code

Section 20.3. Creating Class Definitions from DCDs

Section 20.4. Creating Methods from Interaction Diagrams

Section 20.5. Collection Classes in Code

Section 20.6. Exceptions and Error Handling

Section 20.7. Defining the Sale.makeLineItem Method

Section 20.8. Order of Implementation

Section 20.9. Test-Driven or Test-First Development

Section 20.10. Summary of Mapping Designs to Code

Section 20.11. Introduction to the NextGen POS Program Solution Section 20.12. Introduction to the Monopoly Program Solution Chapter 21. Test-Driven Development and Refactoring

Introduction

Section 21.1. Test-Driven Development

Section 21.2. Refactoring

Section 21.3. Recommended Resources

Part 4. Elaboration Iteration 2 More Patterns

Chapter 22. UML Tools and UML as Blueprint

Introduction

Section 22.1. Forward, Reverse, and Round-Trip Engineering Section 22.2. What is a Common Report of Valuable Features? Section 22.3. What to Look For in a Tool?

Section 22.4. If Sketching UML, How to Update the Diagrams After Coding? Section 22.5. Recommended Resources

Chapter 23. Quick Analysis Update

Introduction

Section 23.1. Case Study: NextGen POS

Section 23.2. Case Study: Monopoly

Chapter 24. Iteration 2More Patterns

Introduction

Section 24.1. From Iteration 1 to 2

Section 24.2. Iteration-2 Requirements and Emphasis: Object Design and Patterns Chapter 25. GRASP: More Objects with Responsibilities

Introduction

Section 25.1. Polymorphism

Section 25.2. Pure Fabrication

Section 25.3. Indirection

Section 25.4. Protected Variations

Chapter 26. Applying GoF Design Patterns

Introduction

Section 26.1. Adapter (GoF)

Section 26.2. Some GRASP Principles as a Generalization of Other Patterns Section 26.3. "Analysis" Discoveries During Design: Domain Model Section 26.4. Factory

Section 26.5. Singleton (GoF)

Section 26.6. Conclusion of the External Services with Varying Interfaces Problem Section 26.7. Strategy (GoF)

Section 26.8. Composite (GoF) and Other Design Principles Section 26.9. Facade (GoF)

Section 26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF) Section 26.11. Conclusion

Section 26.12. Recommended Resources

Part 5. Elaboration Iteration 3 Intermediate Topics

Chapter 27. Iteration 3Intermediate Topics

Introduction

Section 27.1. NextGen POS

Section 27.2. Monopoly

Chapter 28. UML Activity Diagrams and Modeling

Introduction

Section 28.1. Example

Section 28.2. How to Apply Activity Diagrams?

Section 28.3. More UML Activity Diagram Notation

Section 28.4. Guidelines

Section 28.5. Example: NextGen Activity Diagram

Section 28.6. Process: Activity Diagrams in the UP

Section 28.7. Background

Chapter 29. UML State Machine Diagrams and Modeling

Introduction

Section 29.1. Example

Section 29.2. Definitions: Events, States, and Transitions

Section 29.3. How to Apply State Machine Diagrams?

Section 29.4. More UML State Machine Diagram Notation

Section 29.5. Example: UI Navigation Modeling with State Machines Section 29.6. Example: NextGen Use Case State Machine Diagram Section 29.7. Process: State Machine Diagrams in the UP

Section 29.8. Recommended Resources

Chapter 30. Relating Use Cases

Introduction

Section 30.1. The include Relationship

Section 30.2. Terminology: Concrete, Abstract, Base, and Addition Use Cases Section 30.3. The extend Relationship

Section 30.4. The generalize Relationship

Section 30.5. Use Case Diagrams

Chapter 31. More SSDs and Contracts

Introduction

Section 31.1. NextGen POS

Chapter 32. Domain Model Refinement

Introduction

Section 32.1. New Concepts for the NextGen Domain Model Section 32.2. Generalization

Section 32.3. Defining Conceptual Superclasses and Subclasses Section 32.4. When to Define a Conceptual Subclass?

Section 32.5. When to Define a Conceptual Superclass? Section 32.6. NextGen POS Conceptual Class Hierarchies Section 32.7. Abstract Conceptual Classes

Section 32.8. Modeling Changing States

Section 32.9. Class Hierarchies and Inheritance in Software Section 32.10. Association Classes

Section 32.11. Aggregation and Composition

Section 32.12. Time Intervals and Product PricesFixing an Iteration 1 "Error" Section 32.13. Association Role Names

Section 32.14. Roles as Concepts versus Roles in Associations Section 32.15. Derived Elements

Section 32.16. Qualified Associations

Section 32.17. Reflexive Associations

Section 32.18. Using Packages to Organize the Domain Model Section 32.19. Example: Monopoly Domain Model Refinements Chapter 33. Architectural Analysis

Introduction

Section 33.1. Process: When Do We Start Architectural Analysis? Section 33.2. Definition: Variation and Evolution Points

Section 33.3. Architectural Analysis

Section 33.4. Common Steps in Architectural Analysis

Section 33.5. The Science: Identification and Analysis of Architectural Factors Section 33.6. Example: Partial NextGen POS Architectural Factor Table Section 33.7. The Art: Resolution of Architectural Factors Section 33.8. Summary of Themes in Architectural Analysis Section 33.9. Process: Iterative Architecture in the UP

Section 33.10. Recommended Resources

Chapter 34. Logical Architecture Refinement

Introduction

Section 34.1. Example: NextGen Logical Architecture

Section 34.2. Collaborations with the Layers Pattern

Section 34.3. Other Layer Pattern Issues

Section 34.4. Model-View Separation and "Upward" Communication Section 34.5. Recommended Resources

Chapter 35. More Object Design with GoF Patterns

Introduction

Section 35.1. Example: NextGen POS

Section 35.2. Failover to Local Services; Performance with Local Caching Section 35.3. Handling Failure

Section 35.4. Failover to Local Services with a Proxy (GoF) Section 35.5. Designing for Non-Functional or Quality Requirements

Section 35.6. Accessing External Physical Devices with Adapters Section 35.7. Abstract Factory (GoF) for Families of Related Objects Section 35.8. Handling Payments with Polymorphism and Do It Myself Section 35.9. Example: Monopoly

Section 35.10. Conclusion

Chapter 36. Package Design

Introduction

Section 36.1. Package Organization Guidelines

Section 36.2. Recommended Resources

Chapter 37. UML Deployment and Component Diagrams

Section 37.1. Deployment Diagrams

Section 37.2. Component Diagrams

Chapter 38. Designing a Persistence Framework with Patterns Introduction

Section 38.1. The Problem: Persistent Objects

Section 38.2. The Solution: A Persistence Service from a Persistence Framework Section 38.3. Frameworks

Section 38.4. Requirements for the Persistence Service and Framework Section 38.5. Key Ideas

Section 38.6. Pattern: Representing Objects as Tables

Section 38.7. UML Data Modeling Profile

Section 38.8. Pattern: Object Identifier

Section 38.9. Accessing a Persistence Service with a Facade Section 38.10. Mapping Objects: Database Mapper or Database Broker Pattern Section 38.11. Framework Design with the Template Method Pattern Section 38.12. Materialization with the Template Method Pattern Section 38.13. Configuring Mappers with a MapperFactory Section 38.14. Pattern: Cache Management

Section 38.15. Consolidating and Hiding SQL Statements in One Class Section 38.16. Transactional States and the State Pattern Section 38.17. Designing a Transaction with the Command Pattern Section 38.18. Lazy Materialization with a Virtual Proxy

Section 38.19. How to Represent Relationships in Tables

Section 38.20. PersistentObject Superclass and Separation of Concerns Section 38.21. Unresolved Issues

Chapter 39. Documenting Architecture: UML & the N+1 View Model Introduction

Section 39.1. The SAD and Its Architectural Views

Section 39.2. Notation: The Structure of a SAD

Section 39.3. Example: A NextGen POS SAD

Section 39.4. Example: A Jakarta Struts SAD

Section 39.5. Process: Iterative Architectural Documentation Section 39.6. Recommended Resources

Part 6. Special Topics

Chapter 40. More on Iterative Development and Agile Project Management Introduction

Section 40.1. How to Plan an Iteration?

Section 40.2. Adaptive versus Predictive Planning

Section 40.3. Phase and Iteration Plans

Section 40.4. How to Plan Iterations with Use Cases and Scenarios?

Section 40.5. The (In)Validity of Early Estimates

Section 40.6. Organizing Project Artifacts

Section 40.7. You Know You Didn't Understand Iterative Planning When…

Section 40.8. Recommended Resources

Bibliography

Glossary

Inside Front Cover

Inside Back Cover

Index







**Copyright**

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Publisher: John Wait

Editor in Chief: Don O'Hagan

Acquisitions Editor: Paul Petralia

Marketing Manager: Chris Guzikowski

Managing Editor: John Fuller

Project Editor: Julie Nahil

Manufacturing Buyer: Carol Melville

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales

(800) 382-3419

corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales

international@pearsoned.com

Visit us on the Web: www.phptr.com

*Library of Congress Cataloging-in-Publication Data:*

Larman, Craig.

Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development / Craig Larman. 3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-148906-2 (alk. paper)

1. Object-oriented methods (Computer science) 2. UML (Computer science) 3. System analysis. 4. System design. I. Title.

QA76.9.O35 L37 2004

005.1'17--dc22 2004057647

Copyright © 2005 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.

Rights and Contracts Department

One Lake Street

Upper Saddle River, NJ 07458

All products or services mentioned in this book are the trademarks or service marks of their respective companies or organizations.

Quote acknowledgments:

Paul Erdos: From "The Man Who Only Loved Numbers" by Paul Hoffman.

H.G. Wells: Used by permission of A.P. Watt Ltd. On behalf of the Executors of the Estate of H.G. Wells. Text printed in the United States on recycled paper at Courier in Westford, Massachusetts First printing, October 2004

**Dedication**

*For Julie, Haley, and Hannah*

*Thanks for the love and support.*

**

**

**

**Praise for *Applying UML and Patterns***

"This edition contains Larman's usual accurate and thoughtful writing. It is a very good book made even better."

**Alistair Cockburn**, author, *Writing Effective Use Cases* and *Surviving OO Projects*

"People often ask me which is the best book to introduce them to the world of OO design. Ever since I came across it *Applying UML and Patterns* has been my unreserved choice."

**Martin Fowler**, author, *UML Distilled* and *Refactoring*

"This book makes learning UML enjoyable and pragmatic by incrementally introducing it as an intuitive language for specifying the artifacts of object analysis and design. It is a well written introduction to UML and object methods by an expert practitioner."

**Cris Kobryn**, Chair of the *UML* Revision Task Force and *UML* 2.0 Working Group

"Too few people have a knack for explaining things. Fewer still have a handle on software analysis and design. Craig Larman has both."

**John Vlissides**, author, *Design Patterns* and *Pattern Hatching*

**

**

**Contents by Major Topics**

This book introduces a topic incrementally, spread out over chapters as the case studies unfold. That's useful, but it introduces a problem: How can you find most material on a major subject (e.g., OO Design)? The Index is one solution, but fine-grained; this listing provides another.

**Agile Practices**

What is Agile Modeling? 30

What is an Agile UP? 31

Agile Modeling and Lightweight UML Drawing 214

More on Iterative Development and Agile Project Management 673

**Architecture**

Logical Architecture and UML Package Diagrams 197

Architectural Analysis 541

Logical Architecture Refinement 559

Package Design 613

Documenting Architecture: UML & the N+1 View Model 655

**Domain Modeling**

Domain Models 131

What's the Relationship Between the Domain Layer and Domain Model? 206 Domain Model Refinement 507

**GRASP**

GRASP: Designing Objects with Responsibilities 271

GRASP: More Objects with Responsibilities 413

Handling Payments with Polymorphism and Do It Myself 600

Example: Monopoly 607

**GoF Design Patterns**

What are Patterns? 278

Applying GoF Design Patterns 435

More Object Design with GoF Patterns 579

Designing a Persistence Framework with Patterns 625 **Iterative Development**

Iterative, Evolutionary, and Agile 17

Evolutionary Requirements 53

Iteration 1Basics 123

Requirements to DesignIteratively 195

More on Iterative Development and Agile Project Management 673 **OO Analysis**

*See* Domain Modeling and Operation Contracts

**OO Design**

*See* GRASP and GoF Design Patterns

On to Object Design 213

Package Design 613

**Operation Contracts**

Operation Contracts 181

Operation Contracts and Use Case Realizations 326 More SSDs and Contracts 501

**Patterns**

*See* GRASP and GoF Design Patterns

What are Patterns? 278

**Programming**

Mapping Designs to Code 369

Test-Driven Development and Refactoring 385

**Project Management**

*See* Agile Practices and Iterative Development

More on Iterative Development and Agile Project Management 673 **Requirements**

*See* Use Cases

Evolutionary Requirements 53

Other Requirements 101

Requirements to DesignIteratively 195

UML Activity Diagrams and Modeling 477

UML State Machine Diagrams and Modeling 485

**System Sequence Dgms**

System Sequence Diagrams 173

SSDs, System Operations, Interaction Diagrams, and Use Case Realizations 324 More SSDs and Contracts 501

**Testing**

Test-Driven Development and Refactoring 385

**UML**

What is the UML? 11

Applying UML: Use Case Diagrams 89

Domain Models 131

Applying UML: Sequence Diagrams 177

Applying UML: Package Diagrams 201

UML Interaction Diagrams 221

UML Class Diagrams 249

UML Activity Diagrams and Modeling 477

UML State Machine Diagrams and Modeling 485

Use Case Diagrams 499

UML Deployment and Component Diagrams 621

**Unified Process**

What is an Agile UP? 31

Are There Other Critical UP Practices? 33

More on Iterative Development and Agile Project Management 673 **Use Cases**

Use Cases 61

What is the Relationship Between SSDs and Use Cases? 177 What is a Use Case Realization? 322

Relating Use Cases 493

How to Plan Iterations with Use Cases and Scenarios? 676





**Foreword**

Programming is fun, but developing quality software is hard. In between the nice ideas, the requirements or the "vision," and a working software product, there is much more than programming. Analysis and design, defining how to solve the problem, what to program, capturing this design in ways that are easy to communicate, to review, to implement, and to evolve is what lies at the core of this book. This is what you will learn.

The Unified Modeling Language (UML) has become the universally-accepted language for software design blueprints. UML is the visual language used to convey design ideas throughout this book, which emphasizes how developers really apply frequently used UML elements, rather than obscure features of the language.

The importance of patterns in crafting complex systems has long been recognized in other disciplines. Software design patterns are what allow us to describe design fragments, and reuse design ideas, helping developers leverage the expertise of others. Patterns give a name and form to abstract heuristics, rules and best practices of object-oriented techniques. No reasonable engineer wants to start from a blank slate, and this book offers a palette of readily usable design patterns.

But software design looks a bit dry and mysterious when not presented in the context of a software engineering process. And on this topic, I am delighted that for his new edition, Craig Larman has chosen to embrace and introduce the Unified Process, showing how it can be applied in a relatively simple and low-ceremony way. By presenting the case study in an iterative, risk driven, architecture-centric process, Craig's advice has realistic context; he exposes the dynamics of what really happens in software development, and shows the external forces at play. The design activities are connected to other tasks, and they no longer appear as a purely cerebral activity of systematic transformations or creative intuition. And Craig and I are convinced of the benefits of iterative development, which you will see abundantly illustrated throughout.

So for me, this book has the right mix of ingredients. You will learn a systematic method to do Object-Oriented Analysis and Design (OOA/D) from a great teacher, a brilliant methodologist, and an "OO guru" who has taught it to thousands around the world. Craig describes the method in the context of the Unified Process. He gradually presents more sophisticated design patternsthis will make the book very handy when you are faced with real-world design challenges. And he uses the most widely accepted notation.

I'm honored to have had the opportunity to work directly with the author of this major book. I enjoyed reading the first edition, and was delighted when he asked me to review the draft of his new edition. We met several times and exchanged many e-mails. I have learned much from Craig, even about our own process work on the Unified Process and how to improve it and position it in various organizational contexts. I am certain that you will learn a lot, too, in reading this book, even if you are already familiar with OOA/D. And, like me, you will find yourself going back to it, to refresh your memory, or to gain further insights from Craig's explanations and experience.

Happy reading!

*Philippe Kruchten*

*Professor of Software Engineering, University of British Columbia*

*formerly,*

*Rational Fellow and Director of Process Development for the RUP*

*Rational Software*

*Vancouver, British Columbia*

**

**

**Preface**

Thank you for reading this book! If I can answer a question, or for consulting or coaching a team (in OOA/D, UML, modeling, iterative and agile methods) please contact me at www.craiglarman.com.

This is a practical introduction to object-oriented analysis and design (OOA/D), and to related aspects of iterative development. I am grateful that the previous editions were extremely popular worldwide. I sincerely thank all the readers!

Here is how the book will benefit you.

**First**, the use of object technology is widespread, so mastery of OOA/D is critical for you to succeed in the software world.

*design well*

**Second**, if you are new to OOA/D, you're understandably challenged about how to proceed; this book presents a well-defined iterative roadmapan agile approach to the Unified Processso that you can move in a step-by-step process from requirements to code.

*learn a process roadmap*

**Third**, the Unified Modeling Language (UML) has emerged as the standard notation for modeling, so it's useful to be able to apply it skillfully.

*learn UML for modeling*

**Fourth**, design patterns communicate the "best practice" idioms OO design experts apply. You will learn to apply design patterns, including the popular "gang-of-four" patterns, and the GRASP patterns. Learning and applying patterns will accelerate your mastery of analysis and design.

*learn design patterns*

**Fifth**, the structure and emphasis in this book are based on years of experience in education and mentoring thousands of people in the art of OOA/D. It reflects that experience by providing a refined, proven, and efficient approach to learning the subject, so your investment in reading and learning is optimized.

*learn from experience*

**Sixth**, it exhaustively examines two case studiesto realistically illustrate the entire OOA/D process, and goes deeply into thorny details of the problem.

*learn from a realistic study*

**Seventh**, it shows how to map object design artifacts to code in Java. It also introduces test driven development and refactor.

*design to code, with TDD & refactoring*

**Eighth**, it explains how to design a layered architecture and relate the UI layer to domain and technical services layers.

*layered architecture*

**Finally**, it shows you how to design an OO framework and applies this to the creation of a framework for persistent storage in a database.

*design frameworks*

**

**

**Educator and Web Resources**

You may find related articles of interest at www.craiglarman.com.

Hundreds, if not thousands, of teachers use the book worldwide; it's been translated into at least ten languages. At my website there are a variety of educator resources, including all the book figures organized into Microsoft PowerPoint presentations, sample OOA/D PowerPoint presentations, and more. If you're an educator, please contact me for resources.

I am collecting material from existing educators using the book, to share with other educators. **If you have anything to share, please contact me**.





**Intended Audiencean Introduction!**

This book is an *introduction* to OOA/D, related requirements analysis, and to iterative development with the Unified Process as a sample process; it is not meant as an advanced text. It is for the following audience:

Developers and students with some experience in OO programming, but who are newor relatively newto OOA/D.

Students in computer science or software engineering courses studying object technology.

Those with some familiarity in OOA/D who want to learn the UML notation, apply patterns, or who want to deepen their analysis and design skills.





**Prerequisites**

Some prerequisites are assumedand necessaryto benefit from this book:

Knowledge and experience in an object-oriented programming language such as Java, C#, C++, or Python.

Knowledge of fundamental OO concepts, such as class, instance, interface, polymorphism, encapsulation, and inheritance.

Fundamental OO concepts are not defined.





**Java Examples, But …**

In general, the book presents code examples in Java due to its widespread familiarity. However, the ideas presented are applicable to mostif not allobject-oriented technologies, including C#, Python, and so on.





**Book Organization**

The overall strategy in the organization of this book is that analysis and design topics are introduced in an order similar to that of a software development project running across an "inception" phase (a Unified Process term) followed by three iterations (see Figure P.1).

**1.** The inception phase chapters introduce the basics of requirements analysis. **2.** Iteration 1 introduces fundamental OOA/D and how to assign responsibilities to objects.

**3.**

Iteration 2 focuses on object design, especially on introducing some high-use "design patterns."

**4.**

Iteration 3 introduces a variety of subjects, such as architectural analysis and framework design.

**Figure P.1. The organization of the book follows that of a development project.**

[View full size image]





**About the Author**

Craig Larman serves as chief scientist for Valtech, an international consulting and skills transfer company with divisions in Europe, Asia, and North America. He is also author of the best-selling software engineering and iterative, agile development text *Agile and Iterative Development: A Manager's Guide*. He travels worldwide, from Indiana to India, coaching development teams and managers.

Since the mid 1980s, Craig has helped thousands of developers to apply OOA/D, skillful modeling with the UML, and to adopt iterative development practices.

After a failed career as a wandering street musician, he built systems in APL, PL/I, and CICS in the 1970s. Starting in the early 1980safter a full recovery he became interested in artificial intelligence (having little of his own) and built knowledge systems with Lisp machines, Lisp, Prolog, and Smalltalk. He's also worked in organizations that build business systems in Java, .NET, C++, and Smalltalk. He plays bad lead guitar in his very part-time band, the *Changing Requirements* (it used to be called the *Requirements*, but some band members changed...).

He holds a B.S. and M.S. in computer science from beautiful Simon Fraser University in Vancouver, Canada.





**Contact**

Craig can be reached at craig@craiglarman.com and www.craiglarman.com. He welcomes questions from readers and educators, and speaking, mentoring, and consulting enquiries.





**Enhancements to the Previous Edition**

While retaining the same core as the prior edition, this edition is refined in many ways, including:

UML 2

A second case study

More tips on iterative and evolutionary development combined with OOA/D Rewritten with new learning aids and graphics for easier study

New college-educator teaching resources

Agile Modeling, Test-Driven Development, and refactoring

More on process modeling with UML activity diagrams

Guidance on applying the UP in a light, agile spirit, complementary with other iterative methods such as XP and Scrum

Applying the UML to documenting architectures

A new chapter on evolutionary requirements

Refinement of the use case chapters, using the very popular approach of [Cockburn01]



**Acknowledgments**

First, thanks to my friends and colleagues at Valtech, world-class object developers and iterative development experts, who in some way contributed to, supported, or reviewed the book, including Chris Tarr, Tim Snyder, Curtis Hite, Celso Gonzalez, Pascal Roques, Ken DeLong, Brett Schuchert, Ashley Johnson, Chris Jones, Thomas Liou, Darryl Gebert, and many more than I can name.

To Philippe Kruchten for writing the foreword, reviewing, and helping in many ways.

To Martin Fowler and Alistair Cockburn for many insightful discussions on process and design, quotes, and reviews.

To Oystein Haugen, Cris Kobryn, Jim Rumbaugh, and Bran Selic for reviewing the UML 2 material. To John Vlissides and Cris Kobryn for the kind quotes.

To Chelsea Systems and John Gray for help with some requirements inspired by their Java technology ChelseaStore POS system.

To Pete Coad and Dave Astels for their input.

Many thanks to the other reviewers, including Steve Adolph, Bruce Anderson, Len Bass, Gary K. Evans, Al Goerner, Luke Hohmann, Eric Lefebvre, David Nunn, and Robert J. White.

Thanks to Paul Becker at Prentice-Hall for believing the first edition would be a worthwhile project, and to Paul Petralia for shepherding the later ones.

Finally, a special thanks to Graham Glass for opening a door.





**Typographical Conventions**

This is a **new term** in a sentence. This is a *Class* or *method* name in a sentence. This is an author reference [Bob67].





**Production Notes**

The manuscript was created with Adobe FrameMaker. All drawings were done with Microsoft Visio. The body font is New Century Schoolbook. The final print images were generated as PDF using Adobe Acrobat Distiller, from PostScript generated by the Adobe Universal driver. The UML wall sketch photos were cleaned up with ClearBoard for whiteboard photos.





**Part 1: Introduction**

Chapter 1. Object-Oriented Analysis and Design

Chapter 2. Iterative, Evolutionary, and Agile

Chapter 3. Case Studies





**Chapter 1. Object-Oriented Analysis and Design**

*Le temps est un grand professeur, mais malheureusement il tue tous ses élèves (Time is a great teacher, but unfortunately it kills all its pupils.)*

*Hector Berlioz*

| **Objectives**  Describe the book goals and scope.  Define object-oriented analysis and design (OOA/D).  Illustrate a brief OOA/D example.  Overview UML and visual agile modeling. |
| --- |





**1.1. What Will You Learn? Is it Useful?**

What does it mean to have a good object design? This book is a tool to help developers and students learn core skills in object-oriented analysis and design (OOA/D). These skills are essential for the creation of well-designed, robust, and maintainable software using OO technologies and languages such as Java or C#.

[View full size image]



The proverb "owning a hammer doesn't make one an architect" is especially true with respect to object technology. Knowing an object-oriented language (such as Java) is a necessary but insufficient first step to create object systems. Knowing how to "think in objects" is critical!

This is an introduction to OOA/D while applying the Unified Modeling Language (UML) and patterns. And, to iterative development, using an agile approach to the Unified Process as an example iterative process. It is *not* meant as an advanced text; it emphasizes mastery of the fundamentals, such as how to assign responsibilities to objects, frequently used UML notation, and common design patterns. At the same time, mostly in later chapters, the material progresses to some intermediate-level topics, such as framework design and architectural analysis.

**UML vs. Thinking in Objects**

The book is not just about UML. The **UML** is a standard diagramming notation. Common notation is useful, but there are more important OO things to learn especially, how to think in objects. The UML is not OOA/D or a method, it is just diagramming notation. It's useless to learn UML and perhaps a UML CASE tool, but not really know how to create an excellent OO design, or evaluate and improve an existing one. This is the hard and important skill. Consequently, this book is an introduction to object design.

Yet, we need a language for OOA/D and "software blueprints," both as a tool of thought and as a form of communication. Therefore, this book explores how to *apply* the UML in the service of doing OOA/D, and covers frequently used UML.

**OOD: Principles and Patterns**

How should **responsibilities** be allocated to classes of objects? How should objects collaborate? What classes should do what? These are critical questions in the design of a system, and this book teaches the classic OO design metaphor: **responsibility-driven design**. Also, certain tried-and true solutions to design problems can be (and have been) expressed as best-practice principles, heuristics, or **patterns**named problem-solution formulas that codify exemplary design principles.

This book, by teaching how to *apply* patterns or principles, supports quicker learning and skillful use of these fundamental object design idioms.

**Case Studies**

This introduction to OOA/D is illustrated in some **ongoing case studies** that are followed throughout the book, going deep enough into the analysis and design so that some of the gory details of what must be considered and solved in a realistic problem are considered, and solved.

**Use Cases**

OOD (and all software design) is strongly related to the prerequisite activity of **requirements analysis**, which often includes writing **use cases**. Therefore, the case study begins with an introduction to these topics, even though they are not specifically object-oriented.

**Iterative Development, Agile Modeling, and an Agile UP**

Given many possible activities from requirements through to implementation, how should a developer or team proceed? Requirements analysis and OOA/D needs to be presented and practiced in the context of some development process. In this case, an **agile** (light, flexible) approach to the well-known **Unified Process** (UP) is used as the *sample* **iterative development process** within which these topics are introduced. However, the analysis and design topics that are covered are common to many approaches, and learning them in the context of an agile UP does not invalidate their applicability to other methods, such as Scrum, Feature-Driven Development, Lean Development, Crystal Methods, and so on.

| In conclusion, this book helps a student or developer:  Apply principles and patterns to create better object designs.  Iteratively follow a set of common activities in analysis and design, based on an agile approach to the UP as an example.  Create frequently used diagrams in the UML notation.  It illustrates this in the context of long-running case studies that evolve over several iterations. |
| --- |

**Figure 1.1. Topics and skills covered.**

****

**Many Other Skills Are Important!**

This isn't the *Compleate Booke of Software*; it's primarily an introduction to OOA/D, UML, and iterative development, while touching on related subjects. Building software involves myriad other skills and steps; for example, usability engineering, user interface design, and database design are critical to success.



**1.2. The Most Important Learning Goal?**

There are many possible activities and artifacts in introductory OOA/D, and a wealth of principles and guidelines. Suppose we must choose a single practical skill from all the topics discussed herea "desert island" skill. What would it be?

| A critical ability in OO development is to skillfully assign responsibilities to software objects. |
| --- |

Why? Because it is one activity that must be performedeither while drawing a UML diagram or programmingand it strongly influences the robustness, maintainability, and reusability of software components.

Of course, there are other important skills in OOA/D, but *responsibility assignment* is emphasized in this introduction because it tends to be a challenging skill to master (with many "degrees of freedom" or alternatives), and yet is vitally important. On a real project, a developer might not have the opportunity to perform any other modeling activitiesthe "rush to code" development process. Yet even in this situation, assigning responsibilities is inevitable.

Consequently, the design steps in this book emphasize principles of responsibility assignment.

| Nine fundamental principles in object design and responsibility assignment are presented and applied. They are organized in a learning aid called **GRASP** of principles with names such as *Information Expert* and *Creator*. |
| --- |



**1.3. What is Analysis and Design?**

**Analysis** emphasizes an *investigation* of the problem and requirements, rather than a solution. For example, if a new online trading system is desired, how will it be used? What are its functions?

"Analysis" is a broad term, best qualified, as in *requirements analysis* (an investigation of the requirements) or *object-oriented analysis* (an investigation of the domain objects).

**Design** emphasizes a *conceptual solution* (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Design ideas often exclude low-level or "obvious" detailsobvious to the intended consumers. Ultimately, designs can be implemented, and the implementation (such as code) expresses the true and complete realized design.

As with analysis, the term is best qualified, as in *object-oriented design* or *database design*.

Useful analysis and design have been summarized in the phrase *do the right thing (analysis), and do the thing right (design).*

**

****1.4. What is Object-Oriented Analysis and Design?**

During **object-oriented analysis** there is an emphasis on finding and describing the objectsor conceptsin the problem domain. For example, in the case of the flight information system, some of the concepts include *Plane, Flight,* and *Pilot*.

During **object-oriented design** (or simply, object design) there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, a *Plane* software object may have a *tailNumber* attribute and a *getFlightHistory* method (see Figure 1.2).

**Figure 1.2. Object-orientation emphasizes representation of objects.**

[View full size image]



Finally, during implementation or object-oriented programming, design objects are implemented, such as a *Plane* class in Java.







**1.5. A Short Example**

Before diving into the details of iterative development, requirements analysis, UML, and OOA/D, this section presents a bird's-eye view of a few key steps and diagrams, using a simple examplea "dice game" in which software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.

**Define Use Cases**

[View full size image]



Requirements analysis may include stories or scenarios of how people use the application; these can be written as **use cases**.

Use cases are not an object-oriented artifactthey are simply written stories. However, they are a popular tool in requirements analysis. For example, here is a brief version of the *Play a Dice Game* use case:

**Play a Dice Game:** Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

**Define a Domain Model**

[View full size image]



Object-oriented analysis is concerned with creating a description of the domain from the perspective of objects. There is an identification of the concepts, attributes, and associations that are considered noteworthy.

The result can be expressed in a **domain model** that shows the *noteworthy* domain concepts or objects.

For example, a partial domain model is shown in Figure 1.3.

**Figure 1.3. Partial domain model of the dice game.**

****

This model illustrates the noteworthy concepts *Player, Die,* and *DiceGame*, with their associations and attributes.

Note that a domain model is not a description of software objects; it is a visualization of the concepts or mental models of a real-world domain. Thus, it has also been called a **conceptual object model**.

**Assign Object Responsibilities and Draw Interaction Diagrams**

[View full size image]



Object-oriented design is concerned with defining software objectstheir responsibilities and collaborations. A common notation to illustrate these collaborations is the **sequence diagram** (a kind of UML interaction diagram). It shows the flow of messages between software objects, and thus the invocation of methods.

For example, the sequence diagram in Figure 1.4 illustrates an OO software design, by sending messages to instances of the *DiceGame* and *Die* classes. Note this illustrates a common real world way the UML is applied: by sketching on a whiteboard.

**Figure 1.4. Sequence diagram illustrating messages between software objects.**

****

Notice that although in the real world a *player* rolls the dice, in the software design the *DiceGame* object "rolls" the dice (that is, sends messages to *Die* objects). Software object designs and programs do take some inspiration from real-world domains, but they are *not* direct models or simulations of the real world.

**Define Design Class Diagrams**

[View full size image]



In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, a *static* view of the class definitions is usefully shown with a **design class diagram**. This illustrates the attributes and methods of the classes.

For example, in the dice game, an inspection of the sequence diagram leads to the partial design class diagram shown in Figure 1.5. Since a *play* message is sent to a *DiceGame* object, the *DiceGame* class requires a *play* method, while class *Die* requires a *roll* and *getFaceValue* method.

**Figure 1.5. Partial design class diagram.**

****

In contrast to the domain model showing real-world classes, this diagram shows software classes.

Notice that although this design class diagram is not the same as the domain model, some class names and content are similar. In this way, OO designs and languages can support a **lower representational gap** between the software components and our mental models of a domain. That improves comprehension.

**Summary**

The dice game is a simple problem, presented to focus on a few steps and artifacts in analysis and design. To keep the introduction simple, not all the illustrated UML notation was explained. Future chapters explore analysis and design and these artifacts in closer detail.



**1.6. What is the UML?**

To quote:

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems [OMG03a].

The word *visual* in the definition is a key pointthe UML is the de facto standard *diagramming notation* for drawing or presenting pictures (with some text) related to softwareprimarily OO software.

This book doesn't cover all minute aspects of the UML, a large body of notation. It focuses on frequently used diagrams, the most commonly used features within those, and core notation that is unlikely to change in future UML versions.

The UML defines various **UML profiles** that specialize subsets of the notation for common subject areas, such as diagramming Enterprise JavaBeans (with the *UML EJB profile*).

At a deeper levelprimarily of interest to **Model Driven Architecture** (MDA) CASE tool vendorsunderlying the UML notation is the **UML meta-model** that describes the semantics of the modeling elements. It isn't something a developer needs to learn.

**Three Ways to Apply UML**

In [Fowler03] three ways people apply UML are introduced:

**UML as sketch** Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.

**UML as blueprint** Relatively detailed design diagrams used either for 1) reverse engineering to visualize and better understanding existing code in UML diagrams, or for 2) code generation (forward engineering).

| **UML and "Silver Bullet" Thinking**  There is a well-known paper from 1986 titled "No Silver Bullet" by Dr. Frederick Brooks, also published in his classic book *Mythical Man-Month* (20th anniversary edition). Recommended reading! An essential point is that it's a fundamental mistake (so far, endlessly repeated) to believe there is some special tool or technique in software that will make a dramatic order-of-magnitude difference in productivity, defect reduction, reliability, or simplicity. *And tools don't compensate for design ignorance*.  Yet, you will hear claimsusually from tool vendorsthat drawing UML diagrams will make things much better; or, that Model Driven Architecture (MDA) tools based on UML will be the breakthrough silver bullet.  Reality-check time. The UML is simply a standard diagramming notationboxes, lines, etc. Visual modeling with a common notation can be a great aid, but it is hardly as important as knowing how to design and think in objects. Such design knowledge is a very different and more important skill, and is not mastered by learning UML notation or using a CASE or MDA tool. A person not having good OO design and programming skills who draws UML is just drawing bad designs. I suggest the article *Death by UML Fever* [Bell04] (endorsed by the UML creator Grady Booch) for more on this subject, and also *What UML Is and Isn't* [Larman04].  Therefore, this book is an introduction to OOA/D and *applying* the UML to support skillful OO design. |
| --- |

If reverse engineering, a UML tool reads the source or binaries and generates (typically) UML package, class, and sequence diagrams. These "blueprints" can help the reader understand the big-picture elements, structure, and collaborations.

Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding (perhaps also applying UML sketching).

**UML as programming language** Complete executable specification of a software system in UML. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML "programming language." This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

**Agile modeling** emphasizes *UML as sketch*; this is a common way to apply the UML, often with a high return on the investment of time (which is typically short). UML tools can be useful, but I encourage people to also consider an agile modeling approach to applying UML.

*agile modeling p. 30*

**Three Perspectives to Apply UML**

The UML describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a modeling perspective on these. For example, the same UML class diagram notation can be used to draw pictures of concepts in the real world or software classes in Java.

This insight was emphasized in the Syntropy object-oriented method [CD94]. That is, the same notation may be used for three perspectives and types of models (Figure 1.6):

**1.**

**Conceptual perspective** the diagrams are interpreted as describing things in a situation of the real world or domain of interest.

**2.**

**Specification (software) perspective** the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).

**3.**

**Implementation (software) perspective** the diagrams describe software implementations in a particular technology (such as Java).

**Figure 1.6. Different perspectives with UML.**

[View full size image]



We've already seen an example of this in Figure 1.3 and Figure 1.5, where the same UML class diagram notation is used to visualize a domain model and a design model.

In practice, the specification perspective (deferring the target technology, such as Java versus .NET) is seldom used for design; most software-oriented UML diagramming assumes an implementation perspective.

**The Meaning of "Class" in Different Perspectives**

In the raw UML, the rectangular boxes shown in Figure 1.6 are called **classes**, but this term encompasses a variety of phenomenaphysical things, abstract concepts, software things, events, and so forth.[1]

[1] A UML class is a special case of the general UML model element **classifier**something with structural features and/or behavior, including classes, actors, interfaces, and use cases.

A method superimposes alternative terminology on top of the raw UML. For example, in the UP, when the UML boxes are drawn in the Domain Model, they are called **domain concepts** or **conceptual classes**; the Domain Model shows a conceptual perspective. In the UP, when UML boxes are drawn in the Design Model, they are called **design classes**; the Design Model shows a specification or implementation perspective, as desired by the modeler.

To keep things clear, this book will use class-related terms consistent with the UML and the UP, as follows:

**Conceptual class** real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.

**Software class** a class representing a specification or implementation perspective of a software component, regardless of the process or method.

**Implementation class** a class implemented in a specific OO language such as Java. **UML 1 and UML 2**

Towards the end of 2004 a major new release of the UML emerged, UML 2. This text is based on UML 2; indeed, the notation used here was carefully reviewed with key members of the UML 2 specification team.

**Why Won't We See Much UML for a Few Chapters?**

This is not primarily a UML notation book, but one that explores the larger picture of applying the UML, patterns, and an iterative process in the context of OOA/D and related requirements analysis. OOA/D is normally preceded by requirements analysis. Therefore, the initial chapters introduce the important topics of use cases and requirements analysis, which are then followed by chapters on OOA/D and more UML details.



**1.7. Visual Modeling is a Good Thing**

At the risk of stating the blindingly obvious, drawing or reading UML implies we are working more visually, exploiting our brain's strength to quickly grasp symbols, units, and relationships in (predominantly) 2D box-and-line notations.

This old, simple idea is often lost among all the UML details and tools. It shouldn't be! Diagrams help us see or explore more of the big picture and relationships between analysis or software elements, while allowing us to ignore or hide uninteresting details. That's the simple and essential value of the UMLor any diagramming language.





**1.8. History**

The history of OOA/D has many branches, and this brief synopsis can't do justice to all the contributors. The 1960s and 1970s saw the emergence of OO programming languages, such as Simula and Smalltalk, with key contributors such as Kristen Nygaard and especially Alan Kay, the visionary computer scientist who founded Smalltalk. Kay coined the terms *object-oriented programming* and *personal computing,* and helped pull together the ideas of the modern PC while at Xerox PARC.[2]

[2] Kay started work on OO and the PC in the 1960s, while a graduate student. In December 1979at the prompting of Apple's great Jef Raskin (the lead creator of the Mac)Steve Jobs, co-founder and CEO of Apple, visited Alan Kay and research teams (including Dan Ingalls, the implementor of Kay's vision) at Xerox PARC for a demo of the Smalltalk personal computer. Stunned by what he sawa graphical UI of bitmapped overlapping windows, OO programming, and networked PCshe returned to Apple with a new vision (the one Raskin hoped for), and the Apple Lisa and Macintosh were born.

But OOA/D was informal through that period, and it wasn't until 1982 that OOD emerged as a topic in its own right. This milestone came when Grady Booch (also a UML founder) wrote the first paper titled *Object-Oriented Design*, probably coining the term [Booch82]. Many other well known OOA/D pioneers developed their ideas during the 1980s: Kent Beck, Peter Coad, Don Firesmith, Ivar Jacobson (a UML founder), Steve Mellor, Bertrand Meyer, Jim Rumbaugh (a UML founder), and Rebecca Wirfs-Brock, among others. Meyer published one of the early influential books, *Object-Oriented Software Construction*, in 1988. And Mellor and Schlaer published *Object Oriented Systems Analysis*, coining the term *object-oriented analysis*, in the same year. Peter Coad created a complete OOA/D method in the late 1980s and published, in 1990 and 1991, the twin volumes *Object-Oriented Analysis* and *Object-Oriented Design*. Also in 1990, Wirfs-Brock and others described the responsibility-driven design approach to OOD in their popular *Designing Object-Oriented Software*. In 1991 two very popular OOA/D books were published. One described the OMT method, *Object-Oriented Modeling and Design*, by Rumbaugh et al. The other described the Booch method, *Object-Oriented Design with Applications*. In 1992, Jacobson published the popular *Object-Oriented Software Engineering*, which promoted not only OOA/D, but use cases for requirements.

The UML started as an effort by Booch and Rumbaugh in 1994 not only to create a common notation, but to combine their two methodsthe Booch and OMT methods. Thus, the first public draft of what today is the UML was presented as the *Unified Method*. They were soon joined at Rational Corporation by Ivar Jacobson, the creator of the Objectory method, and as a group came to be known as the *three amigos*. It was at this point that they decided to reduce the scope of their effort, and focus on a common diagramming notationthe UMLrather than a common method. This was not only a de-scoping effort; the Object Management Group (OMG, an industry standards body for OO-related standards) was convinced by various tool vendors that an open standard was needed. Thus, the process opened up, and an OMG task force chaired by Mary Loomis and Jim Odell organized the initial effort leading to UML 1.0 in 1997. Many others contributed to the UML, perhaps most notably Cris Kobryn, a leader in its ongoing refinement.

The UML has emerged as the de facto and de jure standard diagramming notation for object oriented modeling, and has continued to be refined in new OMG UML versions, available at www.omg.org or www.uml.org.





**1.9. Recommended Resources**

Various OOA/D texts are recommended in later chapters, in relation to specific subjects, such as OO design. The books in the history section are all worth studyand still applicable regarding their core advice.

A very readable and popular summary of essential UML notation is *UML Distilled* by Martin Fowler. Highly recommended; Fowler has written many useful books, with a practical and "agile" attitude.

For a detailed discussion of UML notation, *The Unified Modeling Language Reference Manual* by Rumbaugh is worthwhile. Note that this text isn't meant for learning how to do object modeling or OOA/Dit's a UML notation reference.

For the definitive description of the current version of the UML, see the on-line *UML Infrastructure Specification* and *UML Superstructure Specification* at www.uml.org or www.omg.org.

Visual UML modeling in an agile modeling spirit is described in *Agile Modeling* by Scott Ambler. See also www.agilemodeling.com.

There is a large collection of links to OOA/D methods at www.cetus-links.org and www.iturls.com (the large English "Software Engineering" subsection, rather than the Chinese section).

There are many books on software patterns, but the seminal classic is *Design Patterns* by Gamma, Helm, Johnson, and Vlissides. It is truly required reading for those studying object design. However, it is not an introductory text and is best read after one is comfortable with the fundamentals of object design and programming. See also www.hillside.net and www.iturls.com (the English "Software Engineering" subsection) for links to many pattern sites.





**Chapter 2. Iterative, Evolutionary, and Agile**

*You should use iterative development only on projects that you want to succeed. Martin Fowler*

| **Objectives**  Provide motivation for the content and order of the book.  Define an iterative and agile process.  Define fundamental concepts in the Unified Process. |
| --- |





**Introduction**

Iterative development lies at the heart of how OOA/D is best practiced and is presented in this book. Agile practices such as Agile Modeling are key to applying the UML in an effective way. This chapter introduces these subjects, and the Unified Process as a relatively popular *sample* iterative method.

[View full size image]



**Iterative and evolutionary development** contrasted with a sequential or "**waterfall**" lifecycleinvolves early programming and testing of a partial system, in repeating cycles. It also normally assumes development starts before all the requirements are defined in detail; feedback is used to clarify and improve the evolving specifications.

We rely on short quick development steps, feedback, and adaptation to clarify the requirements and design. To contrast, waterfall values promoted big up-front speculative requirements and design steps before programming. Consistently, success/failure studies show that the waterfall is strongly associated with the highest failure rates for software projects and was historically promoted due to belief or hearsay rather than statistically significant evidence. Research demonstrates that iterative methods are associated with higher success and productivity rates, and lower defect levels.





**2.1. What is the UP? Are Other Methods**

**Complementary?**

A **software development process** describes an approach to building, deploying, and possibly maintaining software. The **Unified Process** [JBR99] has emerged as a popular *iterative* software development process for building object-oriented systems. In particular, the **Rational Unified Process** or **RUP** [Kruchten00], a detailed refinement of the Unified Process, has been widely adopted.

Because the Unified Process (UP) is a relatively popular iterative process for projects using OOA/D, and because some process must be used to introduce the subject, the UP shapes the book's structure. Also, since the UP is common and promotes widely recognized best practices, it's useful for industry professionals to know it, and students entering the workforce to be aware of it.

The UP is very flexible and open, and encourages including skillful practices from other iterative methods, such as from **Extreme Programming** (**XP**), **Scrum**, and so forth. For example, XP's **test-driven development, refactoring** and **continuous integration** practices can fit within a UP project. So can Scrum's common project room ("war room") and daily Scrum meeting practice. Introducing the UP is not meant to downplay the value of these other methodsquite the opposite. In my consulting work, I encourage clients to understand and adopt a blend of useful techniques from several methods, rather than a dogmatic "my method is better than your method" mentality.

*test-driven development and refactoring p. 385*

The UP combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and well-documented process description.

To summarize, this chapter includes an introduction to the UP for three reasons:

**1.**

The UP is an *iterative* process. Iterative development influences how this book introduces OOA/D, and how it is best practiced.

**2.**

UP practices provide an example *structure* for how to doand thus how to explainOOA/D. That structure shapes the book structure.

**3.**

The UP is flexible, and can be applied in a lightweight and *agile* approach that includes practices from other agile methods (such as XP or Scrum)more on this later.

**3.**

| This book presents an introduction to an agile approach to the UP, but not complete coverage. It emphasizes common ideas and artifacts related to an introduction to OOA/D and requirements analysis. |
| --- |

**What If I Don't Care About the UP?**

The UP is used as an *example* process within which to explore iterative and evolutionary requirements analysis and OOA/D, since it's necessary to introduce the subject in the context of some process.

But the central ideas of this bookhow to think and design with objects, apply UML, use design patterns, agile modeling, evolutionary requirements analysis, writing use cases, and so forthare independent of any particular process, and apply to many modern iterative, evolutionary, and agile methods, such as Scrum, Lean Development, DSDM, Feature-Driven Development, Adaptive Software Development, and more.



**2.2. What is Iterative and Evolutionary Development?**

A key practice in both the UP and most other modern methods is **iterative development**. In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called **iterations**; the outcome of each is a tested, integrated, and executable *partial* system. Each iteration includes its own requirements analysis, design, implementation, and testing activities.

The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as **iterative and incremental development** (see Figure 2.1). Because feedback and adaptation evolve the specifications and design, it is also known as **iterative and evolutionary development**.

**Figure 2.1. Iterative and evolutionary development.**

[View full size image]



Early iterative process ideas were known as spiral development and evolutionary development [Boehm88, Gilb88].

| Example  As an example (not a recipe), in a three-week iteration early in the project, perhaps one hour Monday morning is spent in a kickoff meeting with the team clarifying the tasks and goals of the iteration. Meanwhile, one person reverse-engineers the last iteration's code into UML diagrams (via a CASE tool), and prints and displays noteworthy diagrams. The team spends the remainder of Monday at whiteboards, working in pairs while agile modeling, sketching rough UML diagrams captured on digital cameras, and writing some pseudocode and design notes. The remaining days are spent on implementation, testing (unit, acceptance, usability, …), further design, integration, and daily builds of the partial system. Other activities include demonstrations and evaluations with stakeholders, and planning for the next iteration. |
| --- |

Notice in this example that there is neither a rush to code, nor a long drawn-out design step that attempts to perfect all details of the design before programming. A "little" forethought regarding the design with visual modeling using rough and fast UML drawings is done; perhaps a half or full day by developers doing design work UML sketching in pairs at whiteboards.

The result of each iteration is an executable but incomplete system; it is not ready to deliver into production. The system may not be eligible for production deployment until after many iterations; for example, 10 or 15 iterations.

The output of an iteration is *not* an experimental or throw-away prototype, and iterative development is not prototyping. Rather, the output is a production-grade subset of the final system.

**How to Handle Change on an Iterative Project?**

The subtitle of one book that discusses iterative development is *Embrace Change* [Beck00]. This phrase is evocative of a key attitude of iterative development: Rather than fighting the inevitable change that occurs in software development by trying (unsuccessfully) to fully and correctly specify, freeze, and "sign off" on a frozen requirement set and design before implementation (in a "waterfall" process), iterative and evolutionary development is based on an attitude of embracing change and adaptation as unavoidable and indeed essential drivers.

This is not to say that iterative development and the UP encourage an uncontrolled and reactive "feature creep"-driven process. Subsequent chapters explore how the UP balances the needon the one handto agree upon and stabilize a set of requirements, withon the other handthe reality of changing requirements, as stakeholders clarify their vision or the marketplace changes.

Each iteration involves choosing a small subset of the requirements, and quickly designing, implementing, and testing. In early iterations the choice of requirements and design may not be exactly what is ultimately desired. But the act of swiftly taking a small step, before all requirements are finalized, or the entire design is speculatively defined, leads to rapid feedbackfeedback from the users, developers, and tests (such as load and usability tests).

And this early feedback is worth its weight in gold; rather than *speculating* on the complete, correct requirements or design, the team mines the feedback from realistic building and testing something for crucial practical insight and an opportunity to modify or adapt understanding of the

requirements or design. End-users have a chance to quickly see a partial system and say, "Yes, that's what I asked for, but now that I try it, what I really want is something slightly different."[1] This "yes…but" process is not a sign of failure; rather, early and frequent structured cycles of "yes…buts" are a skillful way to make progress and discover what is of real value to the stakeholders. Yet this is not an endorsement of chaotic and reactive development in which developers continually change directiona middle way is possible.

[1] Or more likely, "You didn't understand what I wanted!"

In addition to requirements clarification, activities such as load testing will prove if the partial design and implementation are on the right path, or if in the next iteration, a change in the core architecture is required. Better to resolve and *prove* the risky and critical design decisions early rather than lateand iterative development provides the mechanism for this.

Consequently, work proceeds through a series of structured build-feedback-adapt cycles. Not surprisingly, in early iterations the deviation from the "true path" of the system (in terms of its final requirements and design) will be larger than in later iterations. Over time, the system converges towards this path, as illustrated in Figure 2.2.

**Figure 2.2. Iterative feedback and evolution leads towards the desired system. The requirements and design instability lowers over time.**

[View full size image]

**Are There Benefits to Iterative Development?**

Yes. Benefits include:

less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods

early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)

early visible progress

early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders

managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps

the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

**How Long Should an Iteration Be? What is Iteration Timeboxing?**

Most iterative methods recommend an iteration length between two and six weeks. Small steps, rapid feedback, and adaptation are central ideas in iterative development; long iterations subvert the core motivation for iterative development and increase project risk. In only one week it is often difficult to complete sufficient work to get meaningful throughput and feedback; more than six weeks, and the complexity becomes rather overwhelming, and feedback is delayed. A very long timeboxed iteration misses the point of iterative development. Short is good.

A key idea is that iterations are **timeboxed**, or fixed in length. For example, if the next iteration is chosen to be three weeks long, then the partial system *must* be integrated, tested, and stabilized by the scheduled datedate slippage is illegal. If it seems that it will be difficult to meet the deadline, the recommended response is to de-scoperemove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date.





**2.3. What About the Waterfall Lifecycle?**

In a **waterfall** (or sequential) lifecycle process there is an attempt to define (in detail) all or most of the requirements before programming. And often, to create a thorough design (or set of smodels) before programming. Likewise, an attempt to define a "reliable" plan or schedule near the startnot that it will be.

| Warning: Superimposing Waterfall on Iterative  If you find yourself on an "iterative" project where most of the requirements are written before development begins, or there is an attempt to create many thorough and detailed specifications or UML models and designs before programming, know that waterfall thinking has unfortunately afflicted the project. It is not a healthy iterative or UP project, regardless of claims. |
| --- |

Research (collected from many sources and summarized in [Larman03] and [LB03]) now shows conclusively that the 1960s and 1970s-era advice to apply the waterfall wasironicallya poor practice for most software projects, rather than a skillful approach. It is strongly associated with high rates of failure, lower productivity, and higher defect rates (than iterative projects). On average, 45% of the features in waterfall requirements are never used, and early waterfall schedules and estimates vary up to 400% from the final actuals.

*feature use research p. 56*

In hindsight, we now know that waterfall advice was based on *speculation* and *hearsay*, rather than evidence-based practices. In contrast, iterative and evolutionary practices are backed by evidencestudies show they are less failure prone, and associated with better productivity and defect rates.

**Guideline: Don't Let Waterfall Thinking Invade an Iterative or UP Project**

I need to emphasize that "waterfall thinking" often incorrectly still invades a so-called iterative or UP project. Ideas such as "let's write all the use cases before starting to program" or "let's do many detailed OO models in UML before starting to program" are examples of unhealthy waterfall thinking incorrectly super imposed on the UP. The creators of the UP cite this misunderstandingbig up-front analysis and modelingas a key reason for its failed adoption [KL01].

**Why is the Waterfall so Failure-Prone?**

There isn't one simple answer to why the waterfall is so failure-prone, but it is strongly related to a key false assumption underlying many failed software projectsthat the specifications are predictable and stable and can be correctly defined at the start, with low change rates. This turns out to be far from accurateand a costly misunderstanding. A study by Boehm and Papaccio showed that a typical software project experienced a 25% change in requirements [BP88]. And this trend was corroborated in another major study of thousands of software projects, with change rates that go even higher35% to 50% for large projectsas illustrated in Figure 2.3 [Jones97].

**Figure 2.3. Percentage of change on software projects of varying sizes. **

These are *extremely* high change rates. What this data showsas any experienced developer or manager is painfully awareis that software development is (on average) a domain of high change and instabilityalso known as the domain of **new product development**. Software is not usually a domain of predictable or mass manufacturinglow-change areas where it is possible and efficient to define all the stable specifications and reliable plans near the start.

Thus, any analysis, modeling, development, or management practice based on the assumption that things are long-term stable (i.e., the waterfall) is fundamentally flawed. *Change* is the constant on software projects. Iterative and evolutionary methods assume and embrace change and adaptation of *partial and evolving* specifications, models, and plans based on feedback.

**The Need for Feedback and Adaptation**

In complex, changing systems (such as most software projects) feedback and adaptation are key ingredients for success.

Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements.

Feedback from tests and developers to refine the design or models.

Feedback from the progress of the team tackling early features to refine the schedule and estimates.

Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration.





**2.4. How to do Iterative and Evolutionary Analysis and Design?**

This introduction may have given the impression that there is no value in analysis and design before programming, but that is a misunderstanding as extreme as thinking that "complete" up front analysis is skillful. There is a middle way. Here's a short *example* (not a recipe) of how it can work on a well-run UP project. This assumes there will ultimately be 20 iterations on the project before delivery:

**1.**

Before iteration-1, hold the first timeboxed requirements workshop, such as exactly two days. Business and development people (including the chief architect) are present.

On the morning of day one, do high-level requirements analysis, such as identifying just the names of the use cases and features, and key non-functional requirements. The analysis will not be perfect.

Ask the chief architect and business people to pick 10% from this high-level list (such as 10% of the 30 use case names) that have a blending of these three qualities: 1) architecturally significant (if implemented, we are forced to design, build, and test the core architecture), 2) high business value (features business really cares about), and 3) high risk (such as "be able to handle 500 concurrent transactions"). Perhaps three use cases are thus identified: UC2, UC11, UC14.

For the remaining 1.5 days, do intensive detailed analysis of the functional and non functional requirements for these three use cases. When finished, 10% are deeply analyzed, and 90% are only high-level.

**2.**

Before iteration-1, hold an iteration planning meeting in which a subset from UC2, UC11, and UC14 are chosen to design, build, and test within a specified time (for example, four week timeboxed iteration). Note that not all of these three use cases can be built in iteration-1, as they will contain too much work. After choosing the specific subset goals, break them down into a set of more detailed iteration tasks, with help from the development team.

**3.**

Do iteration-1 over three or four weeks (pick the timebox, and stick to it).

On the first two days, developers and others do modeling and design work in pairs, sketching UML-ish diagrams at many whiteboards (along with sketching other kinds of models) in a common war room, coached and guided by the chief architect.

Then the developers take off their "modeling hats" and put on their "programming hats." They start programming, testing, and integrating their work continuously over the remaining weeks, using the modeling sketches as a starting point of inspiration, knowing that the models are partial and often vague.

Much testing occurs: unit, acceptance, load, usability, and so forth.

One week before the end, ask the team if the original iteration goals can be met; if

not, de-scope the iteration, putting secondary goals back on the "to do" list.

On Tuesday of the last week there's a code freeze; all code must be checked in, integrated, and tested to create the iteration baseline.

On Wednesday morning, demo the partial system to external stakeholders, to show early visible progress. Feedback is requested.

**4.**

Do the second requirements workshop near the end of iteration-1, such as on the last Wednesday and Thursday. Review and refine all the material from the last workshop. Then pick another 10% or 15% of the use cases that are architecturally significant and of high business value, and analyze them in detail for one or two days. When finished, perhaps 25% of the use cases and non-functional requirements will be written in detail. They won't be perfect.

**5.** On Friday morning, hold another iteration planning meeting for the next iteration. **6.** Do iteration-2; similar steps.

**7.**

Repeat, for four iterations and five requirements workshops, so that at the end of iteration 4, perhaps 80% or 90% of the requirements have been written in detail, but only 10% of the system has been implemented.

Note that this large, detailed set of requirements is based on feedback and evolution, and is thus of much higher quality than purely speculative waterfall specifications.

**8.**

We are perhaps only 20% into the duration of the overall project. In UP terms, this is the end of the **elaboration phase**. At this point, estimate in detail the effort and time for the refined, high-quality requirements. Because of the significant realistic investigation, feedback, and early programming and testing, the estimates of what can be done and how long it will take are much more reliable.

**9.**

After this point, requirements workshops are unlikely; the requirements are stabilizedthough never completely frozen. Continue in a series of three-week iterations, choosing the next step of work adaptively in each iteration planning meeting on the final Friday, re-asking the question each iteration, "Given what we know today, what are the most critical technical and business features we should do in the next three weeks?"

Figure 2.5 illustrates the approach for a 20-iteration project.

**Figure 2.5. A UML sketch of a sequence diagram from a project.** [View full size image]



In this way, after a few iterations of early exploratory development, there comes a point when the team can more reliably answer "what, how much, when."





**2.5. What is Risk-Driven and Client-Driven Iterative Planning?**

The UP (and most new methods) encourage a combination of **risk-driven** and **client-driven** iterative planning. This means that the goals of the early iterations are chosen to 1) identify and drive down the highest risks, and 2) build visible features that the client cares most about.

Risk-driven iterative development includes more specifically the practice of **architecture-centric** iterative development, meaning that early iterations focus on building, testing, and stabilizing the core architecture. Why? Because not having a solid architecture is a common high risk.

| Book Iterations vs. Real Project Iterations  Iteration-1 of the case studies in this book is driven by learning goals rather than true project goals. Therefore, iteration-1 is not architecture-centric or risk-driven. On a real project, we would tackle difficult and risky things first. But in the context of a book helping people learn fundamental OOA/D and UML, that's impracticalwe need to start with problems illustrating basic principles, not the most difficult topics and problems. |
| --- |





**2.6. What are Agile Methods and Attitudes?**

**Agile development** methods usually apply timeboxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage *agility*rapid and flexible response to change.

**Figure 2.4. Evolutionary analysis and designthe majority in early iterations.**

[View full size image]



It is not possible to exactly define **agile methods**, as specific practices vary widely. However, short timeboxed iterations with evolutionary refinement of plans, requirements, and design is a basic practice the methods share. In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-organizing teams, and more.

Example practices from the Scrum agile method include a *common project workroom* and *self organizing teams* that coordinate through a daily stand-up meeting with four special questions each member answers. Example practices from the Extreme Programming (XP) method include

*programming in pairs* and **test-driven development**.

*TDD p. 385*

Any iterative method, including the UP, can be applied in an agile spirit. And the UP itself is flexible, encouraging a "whatever works" attitude to include practices from Scrum, XP, and other methods.

**The Agile Manifesto and Principles**

**The Agile Manifesto**

*Individuals and interactions over processes and tools*

*Working software over comprehensive*

*documentation*

*Customer collaboration over contract negotiation*

*Responding to change over following a plan*

**The Agile Principles**

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face

conversation.

8. Agile processes promote sustainable development.

9. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

10. Continuous attention to technical excellence and good design enhances agility.

11. Simplicitythe art of maximizing the amount of work not doneis essential.

12. The best architectures, requirements, and designs emerge from self-organizing teams.

13. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

7. Working software is the primary measure of

progress.

In 2001 agroup interested in iterative and agile methods (coining the term) met to find common ground. Out of this came the Agile Alliance (www.agilealliance.com) with a manifesto and statement of principles to capture the spirit of agile methods.





**2.7. What is Agile Modeling?**

Experienced analysts and modelers know the *secret of modeling*:

The purpose of modeling (sketching UML, …) is primarily to *understand*, not to document.

*more on agile modeling p. 214*

That is, the very act of modeling can and should provide a way to better understand the problem or solution space. From this viewpoint, the purpose of "doing UML" (which should really mean "doing OOA/D") is *not* for a designer to create many detailed UML diagrams that are handed off to a programmer (which is a very un-agile and waterfall-oriented mindset), but rather to quickly explore (more quickly than with code) alternatives and the path to a good OO design.

This view, consistent with agile methods, has been called **agile modeling** in the book (amazingly called) *Agile Modeling* [Ambler02]. It implies a number of practices and values, including:

Adopting an agile method does not mean avoiding any modeling; that's a misunderstanding. Many agile methods, such as Feature-Driven Development, DSDM, and Scrum, normally include significant modeling sessions. Even the XP founders, from perhaps the most well known agile method with the least emphasis on modeling, endorsed agile modeling as described by Amblerand practiced by many modelers over the years.

The purpose of modeling and models is primarily to support understanding and communication, not documentation.

Don't model or apply the UML to all or most of the software design. Defer simple or straightforward design problems until programmingsolve them while programming and testing. Model and apply the UML for the smaller percentage of unusual, difficult, tricky parts of the design space.

Use the simplest tool possible. Prefer "low energy" creativity-enhancing simple tools that support rapid input and change. Also, choose tools that support large visual spaces. For example, prefer sketching UML on whiteboards, and capturing the diagrams with a digital camera.[2]

[2] Two whiteboard sketching tips: **One**: If you don't have enough whiteboards (and you should have many large ones), an alternative is "whiteboard" plastic cling sheets which cling to walls (with a static charge) to create whiteboards. The main product in North America is Avery Write-On Cling Sheets; the main product in Europe is LegaMaster Magic-Chart. **Two**: Digital photos of whiteboard images are often poor (due to reflection). Don't use a flash, but use a software "whiteboard image clean up" application to improve the images, if you need to clean them (as I did for this book).

This doesn't mean UML CASE tools or word processors can't be used or have no value, but especially for the creative work of discovery, sketching on whiteboards supports quick creative flow and change. The key rule is ease and agility, whatever the

technology.

Don't model alone, model in pairs (or triads) at the whiteboard, in the awareness that the purpose of modeling is to discover, understand, and share that understanding. Rotate the pen sketching across the members so that all participate.

Create models in parallel. For example, on one whiteboard start sketching a dynamic-view UML interaction diagram, and on another whiteboard, start sketching the complementary static-view UML class diagram. Develop the two models (two views) together, switching back and forth.

Use "good enough" simple notation while sketching with a pen on whiteboards. Exact UML details aren't important, as long as the modelers understand each other. Stick to simple, frequently used UML elements.

Know that all models will be inaccurate, and the final code or design differentsometimes dramatically differentthan the model. Only tested code demonstrates the true design; all prior diagrams are incomplete hints, best treated lightly as throw-away explorations.

Developers themselves should do the OO design modeling, for themselves, not to create diagrams that are given to other programmers to implementan example of un-agile waterfall-oriented practices.

**Agile Modeling in this Book: Why the Snapshots of UML Sketches?**

UML-sketch modeling on whiteboards is a practice Iand many developershave enthusiastically coached and practiced for years. Yet most of the UML diagrams in this book give the impression I don't work that way, because they've been drawn neatly with a tool, for readability. To balance that impression the book occasionally includes digital snapshot pictures of whiteboard UML sketches. It sacrifices legibility but reminds that agile modeling is useful and is the actual practice behind the case studies.

For example, Figure 2.5 is an unedited UML sketch created on a project I was coaching. It took about 20 minutes to draw, with four developers standing around. We needed to understand the inter-system collaboration. The act of drawing it together provided a context to contribute unique insights and reach shared understanding. This captures the feel of how agile modelers apply the UML.



**2.8. What is an Agile UP?**

The UP was not meant by its creators to be heavy or un-agile, although its large *optional* set of activities and artifacts have understandably led some to that impression. Rather, it was meant to be adopted and applied in the spirit of adaptability and lightnessan **agile UP**. Some examples of how this applies:

Prefer a *small* set of UP activities and artifacts. Some projects will benefit more than others, but, in general, keep it simple. Remember that all UP artifacts are optional, and avoid creating them unless they add value. Focus on early programming, not early documenting.

*customizing UP p. 37*

Since the UP is iterative and evolutionary, requirements and designs are not completed before implementation. They adaptively emerge through a series of iterations, based on feedback.

*evolutionary A&D p. 25*

Apply the UML with agile modeling practices.

*agile models p. 30*

There isn't a *detailed* plan for the entire project. There is a high-level plan (called the **Phase Plan**) that estimates the project end date and other major milestones, but it does not detail the fine-grained steps to those milestones. A detailed plan (called the **Iteration Plan**) only plans with greater detail one iteration in advance. Detailed planning is done adaptively from iteration to iteration.

*agile PM p. 673*

The case studies emphasize a relatively small number of artifacts, and iterative development, in the spirit of an agile UP.



**2.9. Are There Other Critical UP Practices?**

The central idea to appreciate and practice in the UP is short timeboxed iterative, evolutionary, and adaptive development. Some additional best practices and key concepts in the UP:

tackle high-risk and high-value issues in early iterations

continuously engage users for evaluation, feedback, and requirements

build a cohesive, core architecture in early iterations

continuously verify quality; test early, often, and realistically

apply use cases where appropriate

do some visual modeling (with the UML)

carefully manage requirements

practice change request and configuration management



**2.10. What are the UP Phases?**

A UP project organizes the work and iterations across four major phases:

**1. Inception** approximate vision, business case, scope, vague estimates.

**2.**

**Elaboration** refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.

**3.**

**Construction** iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.

**4. Transition** beta tests, deployment.

These phases are more fully defined in subsequent chapters.

This is *not* the old "waterfall" or sequential lifecycle of first defining all the requirements, and then doing all or most of the design.

Inception is not a requirements phase; rather, it is a feasibility phase, where just enough investigation is done to support a decision to continue or stop.

Similarly, elaboration is not the requirements or design phase; rather, it is a phase where the core architecture is iteratively implemented, and high-risk issues are mitigated.

Figure 2.6 illustrates common schedule-oriented terms in the UP. Notice that one development cycle (which ends in the release of a system into production) is composed of many iterations.

**Figure 2.6. Schedule-oriented terms in the UP.**

[View full size image]



**2.11. What are the UP Disciplines?**

The UP describes work activities, such as writing a use case, within **disciplines**a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis. In the UP, an **artifact** is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on.

There are several disciplines in the UP; this book focuses on some artifacts in the following three:

**Business Modeling** The Domain Model artifact, to visualize noteworthy concepts in the application domain.

**Requirements** The Use-Case Model and Supplementary Specification artifacts to capture functional and non-functional requirements.

**Design** The Design Model artifact, to design the software objects.

A longer list of UP disciplines is shown in Figure 2.7.

**Figure 2.7. UP disciplines.**

[View full size image]



In the UP, **Implementation** means programming and building the system, not deploying it. The **Environment** discipline refers to establishing the tools and customizing the process for the projectthat is, setting up the tool and process environment.

**What is the Relationship Between the Disciplines and Phases?**

As illustrated in Figure 2.7, during one iteration work goes on in most or all disciplines. However, the relative effort across these disciplines changes over time. Early iterations naturally tend to apply greater relative emphasis to requirements and design, and later ones less so, as the requirements and core design stabilize through a process of feedback and adaptation.

Relating this to the UP phases (inception, elaboration, …), Figure 2.8 illustrates the changing relative effort with respect to the phases; please note these are suggestive, not literal. In elaboration, for example, the iterations tend to have a relatively high level of requirements and design work, although definitely some implementation as well. During construction, the emphasis is heavier on implementation and lighter on requirements analysis.

**Figure 2.8. Disciplines and phases.**

[View full size image]



**How is the Book Structure Influenced by UP Phases and Disciplines?** With respect to the phases and disciplines, what is the focus of the case studies?

| The case studies emphasize the inception and elaboration phase. They focus on some artifacts in the Business Modeling, Requirements, and Design disciplines, as this is where requirements analysis, OOA/D, patterns, and the UML are primarily applied. |
| --- |

The earlier chapters introduce activities in inception; later chapters explore several iterations in elaboration. The following list and Figure 2.9 describe the organization with respect to the UP phases.

**1.** The inception phase chapters introduce the basics of requirements analysis. **2.** Iteration 1 introduces fundamental OOA/D and assignment of responsibilities to objects. **3.** Iteration 2 focuses on object design, especially on introducing some high-use "design

**4.**