Design patterns are immensely powerful, but to build large-scale robust systems, you need more. Pattern-Oriented Analysis and Design introduces a methodology for "composing" proven design patterns into reliable, robust large-scale software systems. Using POAD, you can quickly build systems that are far more robust, scalable, and maintainable-using UML class diagrams as your building blocks.

- POAD: What it is, how it works, and what problems it solves
- Structural and behavioral approaches to design pattern composition
- Design models and UML techniques for pattern composition
- POAD processes: in depth coverage of analysis, design, and design refinement
- A systematic process to compose design patterns
- 4 chapter-length case studies: feedback control, customer behavior simulation, digital content processing, and distributed medical informatics
- Building on POAD: advanced trends and research

Pattern-Oriented Analysis and Design takes design patterns to the next level. Whether you're an architect, designer, developer, or manager, it will help you build better software systems faster.
Table of Contents

Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems

By Sherif M. Yacoub, Hany H. Ammar

Publisher: Addison Wesley
Pub Date: August 28, 2003
ISBN: 0-201-77640-5
Pages: 416

Copyright
Preface
What Do We Reuse?
Composing Design Patterns
POAD
Audience
Foreword
Part I. Introduction
  Chapter 1. Pattern-Oriented Analysis and Design
    The Role of Patterns in Software Development
    Purpose of POAD
    Pattern-Oriented Design Issues
    POAD Is a Solution
    What Is Covered in this Book?
  Chapter 2. Design Patterns and Software Engineering
    Design Patterns in the Software Lifecycle
    POAD and OO Technology
    Design Patterns
    Design Frameworks
    Design Pattern Composition Approaches
    Summary
  Chapter 3. Composition of Design Patterns
    Behavioral Composition Techniques
    Structural Composition of Design Patterns
    Further Reading
Part II. Technological Aspects of POAD
Chapter 4. Constructional Design Patterns as Components

Which Patterns to Use in POAD
Software Components
Specifying a Pattern as a Component
Component Interfaces
Interface Properties
Pattern Interfaces
Summary

Chapter 5. Visual Design Models
Pattern Composition Models
Pattern-Level Model
Pattern-Level with Interfaces Model
Detailed Pattern-Level Model
Characteristics of the POAD Design Models
Summary

Chapter 6. UML Support for Design Patterns
Patterns as Mechanisms
Architectural Patterns
Patterns as Packages
Patterns and Components
Modeling Pattern-Oriented Designs
A Comparison
Summary

Part III. Process Aspects of POAD
Chapter 7. POAD: The Process
Stringing Versus Overlapping
POAD Process Outline (The Nutshell)
POAD and Code Generation
POAD Characteristics
Pattern-Oriented Frameworks
Benefits and Limitations
Summary

Chapter 8. Analysis Phase
Overview
Requirements Analysis
Acquaintance
Retrieval
Pattern Selection
Summary

Chapter 9. Design Phase
Overview
Constructing Pattern-Level Models
Constructing Pattern-Level with Interfaces Models
Constructing Detailed Pattern-Level Models
Summary

Chapter 10. Design-Refinement Phase
Overview
Instantiating Pattern Internals
Developing the Initial Class Diagram
Design Optimization
Using POAD for Developing Frameworks

Summary

Part IV. Case Studies

Chapter 11. Feedback Control Systems

POAD Analysis for the Feedback Control Framework
POAD Design for the Feedback Control Framework
POAD Design Refinement for the Feedback Control Framework
Example: Quality Control in Production Lines
Sample Implementation
Summary

Chapter 12. Simulation of Waiting Queues

Background and Requirements
POAD Analysis for the Waiting-Queues Simulation Architecture
POAD Design for the Waiting-Queues Simulation Architecture
POAD Design Refinement for the Waiting-Queues Simulation Architecture
Sample Implementation
Summary

Chapter 13. A Digital Content Remastering Application

Document Understanding
Pattern-Oriented Analysis and Design for the Distribution Subsystem
Pattern-Oriented Analysis and Design for the Filtering Subsystem
Summary

Chapter 14. A Medical Informatics System

Medical Informatics Standards
Digital Imaging and Communication in Medicine (DICOM)
Pattern-Oriented Analysis and Design of the DICOM UL
Pattern-Oriented Analysis and Design for the Client Application Entity
Summary

Part V. Automation and Summary

Chapter 15. Relation to UML Metamodel and Specification

Syntax Is Not Enough
The Object Constraint Language
Adding POAD Constructs to the UML Metamodel
Abstract Syntax
Well-Formedness Rules
Semantics
Summary

Chapter 16. Tool Support for Designing with Patterns

The Need for Tool Support
Pattern Tools
Requirements and Features of a POAD Tool
Summary

Chapter 17. Wrapping Up

Systematic Composition of Design Patterns
POAD Characteristics
POAD and Software Reuse
Applying POAD
Pattern Literature
Future Trends in Pattern Composition
Conclusion
Appendix A. Interfaces for Constructional Design Patterns
Strategy Pattern
Observer Pattern
Composite Pattern
Reactor Pattern
Template_method Pattern
Proxy Pattern
Abstract Factory Pattern
Builder Pattern
Mediator Pattern
Command Pattern

Appendix B. State of the Art and Practice in Design Patterns
Pattern Mining
Pattern Description Formats (Templates)
Tool Support
Pattern Formalism
Pattern Languages
Pattern-Based Implementation
More Challenges

Glossary
Bibliography
Dedication

To my mother Zienab Halawa, and the memory of my father Mohamed Yacoub
  To my sister Noha Yacoub and my brother Yasser Yacoub
  To my uncle Samir Halawa
  To my professors Hany Ammar and Ali Mil

Sherif Yacoub

To my dear mother Lila Awny, and the memory of my late father Hussein Ammar,
  To my wife Mona, my children Kareem, Taher, Hussein, Mariam, and Hoda,
  To my sisters Azza and Hala, and to my aunts Ragia and Samih
  To the Engineering Faculty of the International Islamic
    University Malaysia, where the background on
    Design patterns caught my attention
  During my leave at IIUM
  In the spring of
  1995

Hany Ammar
Preface

The most difficult part of building software is not coding; it is the decisions you make early at the design level. Those design decisions live with the system for the rest of its lifetime.

Although this statement might offend many software developers who are strong believers in coding and implementation, in reality it should not. It is more a compliment than an insult! How?

We believe that design is a critical phase of the software development lifecycle. Good design decisions eventually result in a good product, and bad design decisions generally affect the quality of the final product. But the question is, How do we make a good design decision, and how can we assess that such a decision is “good” when we do not have the final product to test our decisions early at the design stage?

Software engineering disciplines realized this paradox a long time ago. The shift from a waterfall development lifecycle to an iterative and rapid prototype lifecycle is a solid proof. In any organization, prototyping is a well-appreciated practice. When you do not know whether your idea is implementable, try to implement a reduced version and test it. When we do so, we learn things at the late implementation and coding stage that we were not aware of at the design stage.

Wouldn’t it be great to have someone we trust to whom we can explain the problem and get some useful answers like, such as “I have implemented this before; it does not work because…” or “I have implemented it before, and this is the way to do it. Moreover, this implementation has these advantages and these drawbacks.”

Design patterns were introduced to serve as the advice from the expert. The real power behind patterns is that they are abstractions from the real world. Experienced software designers and developers have implemented and tested solutions to recurring design problems. Design patterns capture their experiences and present them to all other designers in a form that defines what problem is being solved, how it is solved, why the solution is good, and the implications of using that solution.

Therefore, design patterns are captured by experienced software developers and designers during implementation of many applications, then documented at the design phase (possibly with some implementation examples). Other designers then use and deploy these patterns in developing new applications.

The design decisions that we make at the design phase are crucial to the application development. We need to make good design decisions. In reality, we cannot make good design decisions unless we have the experience that enables us to make these decisions. Experienced software developers and designers have this experience, and they convey it to us in the form of design patterns. Hence, software developers make a major contribution to the design process by documenting and presenting those patterns in a form that is usable at the design stage. They make life easier for designers by providing them with these bulletproof, good design solutions. Whereas we believe that design is a critical phase in software development, we also believe that experienced developers and designers have so much to contribute to make the design process a success.
What Do We Reuse?

Reusing software is one approach to expedite the software development process. The question is, then, what can we reuse and how? Code is the most common form of reuse. Before developing a software component, we actively browse the Internet for open source code that we can borrow, modify, and reuse. Reusing designs is a less frequent practice than reusing code due to the complexity and difficulty of constructing generic designs and instantiating them. Moreover, code is more tangible than design, since we can deploy and execute code with little or no modifications. However, it is very unlikely that we will find a black box component to satisfy all our requirements. It is also very risky to modify the source code (if it is available), since this may break the component integrity and the functionality for which it was originally built. Therefore, many software developers prefer to reuse the idea of the solution and have it implemented their way. Designs are presented at higher levels in the form of design models that require further instantiation and implementation. Design patterns help in leveraging the reuse level to the design phase by providing the design models (and sample implementations) that can be reused.
Composing Design Patterns

When we browse existing work and literature on design patterns, we realize that most of the effort is expended in discovering and documenting patterns, and little work is concerned with systematically applying these reusable designs in developing new applications. The problem that deserves more attention is how to compose design patterns to develop software and how these composition approaches are supported by versatile design models such as the Unified Modeling Language.

We generally classify approaches to design applications using patterns as follows:

1. **Incidental or ad hoc.** A design pattern provides a solution together with the forces and consequences of applying this solution. However, this is not usually sufficient to systematically develop with patterns. For instance, the coincidental use of a Strategy pattern in the implementation of a control application is not a systematic approach to deploy patterns. This is simply because there is no process to guide the development and to integrate the pattern with other design artifacts. Hence, the design process is not repeatable.

2. **Systematic.** A systematic approach to design with patterns goes beyond just applying a certain pattern. Systematic approaches can be classified as
   a. **Pattern Languages.** A pattern language provides a set of patterns that solve problems in a specific domain. Pattern languages not only provide the patterns themselves but also the relationships between these patterns. They imply the process to apply the language to completely solve a specific set of design problems.
   b. **Development processes.** A systematic development process defines a pattern composition approach, analysis and design steps, design models, and tools to automate the development steps.

We advocate a systematic development processes for developing with patterns, since this is the only way to make design patterns a common practice in software development. To improve the practice of systematically deploying design patterns in developing software, we need to

- Define composition techniques that can be used to construct applications by composing design patterns, and
- Support these composition techniques with appropriate modeling languages and views.
POAD

While a great deal of research and practice has been devoted to discovering new design patterns, very little has been concerned with the systematic process of "gluing" and "composing" design patterns to develop software applications. This book specifically addresses this problem and provides a practical methodology to compose and deploy design patterns.

This book presents an approach to design software applications using design patterns. It describes a POAD methodology that produces pattern-oriented designs. POAD takes a structural composition approach to glue patterns at the high-level design. It uses the notion of constructional design patterns as design components with interfaces.

POAD is based on the premise that at some design level, it is sufficient to know that some patterns are used in the application, and it is not necessary to overwhelm the designer with the details of the internal design of each pattern. Wouldn't it be nice to work at a higher level than class diagrams and yet know that elements at that level have well-proven class diagrams? This is achieved by POAD. POAD provides logical views to represent the application design as a composition of patterns and provides the necessary means to trace participants of those patterns into the application's final class diagram.

The book provides a briefing of existing design pattern composition approaches and then describes an example-driven methodology to develop robust software designs using patterns as their building blocks. The book describes the technological aspects and the process aspects of the methodology. The technological aspects focus on the models required to glue design patterns together, and the process aspects walk the designers and architects through the various analysis and design steps.
Audience

The intended audiences for this book include the following:

1. Software architects and software designers seeking illustrative techniques to deploy patterns in designing software applications. They will learn how to construct robust, maintainable software architectures using design patterns as their building blocks.

2. Practitioners seeking state of the art and practice in applying design patterns and learning about using pattern catalogs to build software.

3. Application developers seeking benefits from applying design patterns early at the design level rather than applying them only at the code level. The case studies described in the book give hands-on experience in applying design patterns. The examples in Part IV provide useful illustrations.

4. Computer science and software engineering students learning about using design patterns as a good software engineering practice in designing applications. The case studies help students understand and apply basic design patterns to develop applications.

5. Researchers seeking state of the art in pattern composition and an understanding of related issues that could be topics of future research initiatives. The book helps researchers unveil the research problems in design pattern composition.

6. The book helps professors and lecturers in preparing design patterns courses, case studies, and projects by serving as a reference book for pattern composition approaches as well as simple and complex case studies.

7. Reuse managers who want to learn how reusing design patterns can be useful in building a robust, maintainable software architecture. The book helps organization managers in adopting design pattern reuse programs by illustrating an easy-to-apply methodology that can be used early in the software development process.

Prior Knowledge

The following is the background required for the audience:

- Knowledge of the Unified Modeling Language, specifically class and package diagram models.
- Basic OO design concepts, including inheritance, delegation, aggregation, and so on.
- Knowledge of the basic concepts of design patterns, including what patterns are and familiarity with some pattern examples from any pattern catalog book.

Scope

This book is not about

- Teaching OO design models such as the UML.
Teaching the basics of design patterns.

Documenting new design patterns.

This book is about

Deploying design patterns in software development.

Composing design patterns.

How to Read the Book

Part I introduces POAD concepts. Chapter 1 is an introduction to the POAD methodology. In Chapter 1 we discuss the type of problems that POAD solves. Chapter 2 discusses the role of patterns in software design. In Chapter 3 we classify design pattern composition approaches into structural and behavioral composition mechanisms. Chapter 3 elaborates on the behavioral and structural composition approaches respectively and discusses examples from existing design composition techniques. The chapter contains references to further readings on these composition mechanisms.

Part II discusses the technological aspects of POAD. In Chapter 4 we discuss the role of design patterns as building blocks of software design and which design patterns can be used with POAD. Chapter 5 introduces the design models that we use to compose design patterns. It also shows how the UML syntactically supports these models. In Chapter 6 we discuss the UML support for design patterns. We compare different UML approaches to model design patterns and their composition.

Part III discusses the process aspects of POAD. Chapter 7 describes the procedures to apply POAD in the design of software systems. We first discuss the stringing and overlapping pattern composition approaches, then illustrate how POAD reaps the benefits of the two worlds. This chapter also summarizes the analysis, design, and design-refinement phases of POAD and illustrates the overall outline of the process. Chapters 8, 9, and 10 elaborate more on the analysis, design, and design-refinement phases respectively and discuss the development and modeling steps within each phase.

Part IV provides case studies and illustrates the application of the POAD methodology to develop pattern-oriented designs and frameworks. We show examples of applying the methodology to four case studies.

Chapter 11 illustrates the application of POAD in the development of a pattern-oriented design framework for feedback control systems as an example of reactive systems. The framework is generic and is easily instantiable in developing application-specific control systems.

Chapter 12 illustrates the application of POAD in the development of a pattern-oriented design for the domain of simulation of waiting queues as an example of a product line. This framework deals with simulation of customers lining up for service from one or more service stations, such as the supermarket checkout counter or a self-serve car wash.

Chapter 13 illustrates the application of POAD in the development of a pattern-oriented design for the domain of digital-content processing and manipulation. This application is used to read, process, and handle digital content where heterogeneous source and delivery channels are supported, digital media is converted from one format to another, and metadata is extracted.

Chapter 14 illustrates the application of POAD in the development of a pattern-oriented design for part of a distributed medical informatics system that is based on the Digital Imaging and Communication in Medicine (DICOM) standard.

Part V discusses the automation of POAD and wraps up the discussion. Chapter 16 discusses the metamodeling support of UML semantics to POAD models. In Chapter 16 we discuss the tool support for applying, modeling, and composing design patterns. Chapter 17 discusses possible future trends that could build on top of the POAD methodology. Appendix A describes the pattern interfaces for some patterns that are used in the case studies in Part 4. Appendix B contains a discussion about the state of the art and practice in design patterns. The glossary provides the collection of terms used in the book, and finally a large selection of bibliographic information related to using patterns in software development is presented.

Sherif Yacoub and Hany Ammar
January 2003
Foreword

With the growing demand on rapid software development—to meet time-to-market needs—software development processes are shifting from the traditional development starting from scratch to reuse of existing solutions, whenever possible. With increases in the complexity of software systems, development from scratch has simply become an obsolete alternative. The question then becomes: what can we reuse and how can we reuse it? This book provides an answer to these questions: you can reuse design ideas and models in terms of design patterns, and you can reuse them in a systematic process called Pattern-Oriented Analysis and Design (POAD).

Design patterns and application frameworks are two essential technologies in developing complex software systems that are maintainable and stable (see Building Application Frameworks: Object-Oriented Foundations of Framework Design, Fayad et.al. 1999). Although black box component reuse and code reuse are effective techniques, practical experience shows that design reuse is equally if not more important. Design and architecture ideas stay with the system for its lifetime and hence an initial stable design increases chances of building a successful system.

No single pattern is used in isolation! A software system is a composition of multiple patterns and frameworks that are sometimes domain-specific and sometimes generic. Over the last decade, hundreds of software design patterns have been extracted from successful projects and documented. On the other hand, integrating those patterns together to develop application designs is far from complete. Application designers need design models that capture pattern compositions; they need systematic processes to guide them throughout the development process.

POAD is one successful methodology that provides a complete solution for composing design patterns. This book describes the detailed methodology. The key strengths of this book are:

- **Composition models**: it uses UML design models—mainly class and package diagrams—to illustrate how patterns can be glued together to create a sound design.
- **Processes**: it describes what the designer needs to accomplish at each phase—and hence provides a unique systematic approach to compose design patterns.
- **Case studies**: it illustrates POAD in action by describing four examples, including code samples.

The traceability among various designs models at different levels of abstraction—and linking all byproducts of the composition process—is another key differentiator in POAD. It is the practicality of the approach and its ease-of-use that makes it viable in real world application development.

Read carefully, try the examples, and apply them in your application designs. You will find it rewarding! Enjoy!

MOHAMED FAYAD
Professor, Computer Engineering, San Jose State University
CEO, ActiveFrameworks, Inc.
Part I: Introduction

In Part I we review design patterns and lay the groundwork for the Pattern-Oriented Analysis and Design (POAD) methodology. Chapter 1 introduces the purpose of the POAD methodology: why it is needed and how it is useful. Chapter 2 discusses the role that design patterns play in software engineering in general and object-oriented design in particular. Most of the experiences in using patterns follow an ad hoc approach for pattern instantiation and composition. Systematic development using patterns utilizes a composition mechanism to glue patterns together at the design level. Generally, we categorize composition mechanisms as behavioral and structural composition. We discuss some well-known behavior-based and structure-based pattern composition approaches in Chapter 3.
Chapter 1. Pattern-Oriented Analysis and Design

The Role of Patterns in Software Development

Purpose of POAD

Pattern-Oriented Design Issues

POAD Is a Solution

What Is Covered in this Book?
The Role of Patterns in Software Development

As the complexity of software systems increases, we look for approaches to facilitate the development of software applications. Design patterns [Gamma et al. 1995; Buschmann et al. 1996] and design frameworks [Johnson & Foote 1988; Fayad & Schmidt 1997; Pree 1996; Fayad & Schmidt 1999] are among these promising approaches. Design patterns promise reuse benefits early in the development lifecycle. To reap the benefits of deploying these proven design solutions, we need to define design composition techniques to construct applications using patterns. Versatile design models should be developed to support these techniques.

Reusing software in practical applications is a difficult task, yet it is essential to reduce the development effort and assure higher software quality. Design patterns help in leveraging reuse to the design phase because they provide a common vocabulary for design, they provide a means of understanding designs, and they are proven building blocks from which more complex applications are built. The collection of widely available pattern catalogues stimulates further thinking on how to use these trusted solutions to develop applications. Experienced designers and researchers have expended much effort in documenting good quality practices in software design as design patterns. Whereas a lot of attention is given to finding and documenting design patterns, techniques to deploy and glue these proven design solutions are still lacking systematic support.

Designing applications by systematically deploying design patterns is not a trivial process. Although approaches for design using pattern composition techniques have been proposed, they fall short of the goal of having a systematic process. The purpose of this book is exactly to achieve this goal.
Purpose of POAD

As the demand on software systems increases, researchers as well as practitioners look for development methodologies and techniques to automate the production of software and facilitate its maintainability. These techniques have recently embodied design patterns and frameworks. In particular, we recognize the need for a development methodology to develop large-scale complex systems and, at the same time, learn from the experiences of other systems designers in solving recurring design problems.

The documentation of design patterns, as it stands, describes details about a pattern, its usage, structure, behavior of participants, forces and consequences, and guidelines for implementation. What we perceive missing is how to compose these patterns together to develop applications. A complete system cannot, nor will ever, be built from a single pattern. It is the integration and composition of patterns that makes a whole system.

We can compose patterns together at the class level or the object level. Class models expose the implementation and maintenance aspects of a pattern, while object models expose the runtime, behavioral, and role aspects. Several researchers and practitioners [Reenskaug 1996; Riehle 1997] address the problem of gluing patterns using role and responsibility modeling (behavioral composition). Less attention is given to the problem of composing patterns as a composition of classes (structural composition). This book is an advancement of the techniques in the structural composition category. We believe that patterns are not only about behaviors or roles; their structure aspects are the ones that we implement using traditional object-oriented programming languages. Though it is easier to understand the relationship between objects using behavior models, it is easier to implement that behavior if we have a structure model that captures the classes and their relationships. It is also more likely that developers and designers will continue to use OO programming languages such as Java and C++, and it is less likely that they will use other nonstandard programming construct. Therefore, a pattern composition approach that uses structure models, which have one-to-one mappings to program constructs such as class models, is needed.

The purpose of Pattern-Oriented Analysis and Design (POAD) is to

- **Promote pattern-based development.** We are looking for ways to get more designers to use patterns. We want to attract novice designers and help them use patterns by providing simplified approaches and methods that they can follow to use patterns in their design process. To promote pattern-based development, we need to define composition approaches that are easy to use.

- **Develop systematic approaches to glue patterns.** There is an increasing need to develop systematic composition approaches that facilitate the process of gluing patterns. Models that facilitate the integration of design patterns at the design level should be developed to support these approaches.

- **Develop design frameworks.** We can facilitate the development of design frameworks by using patterns as design building blocks.

- **Improve design quality.** Design patterns are good quality designs. Reusing patterns in a design is anticipated to improve the design quality of software applications built using patterns as their basic building blocks.
Pattern-Oriented Design Issues

In promoting pattern-based development and creating new approaches to compose patterns, we are confronted with many challenges:

- **What qualifies a pattern as a design component?** To use patterns as building blocks, we need to find the characteristics that qualify a pattern as a design component. [1] How can we define pattern interfaces for the purpose of integration with other patterns?

  [1] More elaborate discussion about the differences between design components and deployable components will follow in Part II.

- **Can we compose applications solely from design patterns?** Many applications use one or more patterns in their design. The challenge is whether applications can be built by gluing design patterns? How can these patterns interface? What are pattern interfaces, and what interface mismatch issues could arise? Given the current literature of design patterns, is this repository sufficient to design applications using design patterns? What type of patterns can be used?

- **How can we systematically develop applications using design patterns?** Is there a well-defined design process that can be followed to develop applications using design patterns as their building blocks?

Figure 1-1. Can we compose application designs using design patterns?
POAD Is a Solution

We developed POAD to address the above problems. In POAD, patterns are stringed at the high-level design, and their constituents are merged in the subsequent design stages to obtain a dense and profound design. The POAD approach provides the following solutions:

1. **Modeling patterns as design components.** Since patterns will be the building blocks of pattern-oriented designs, they should first possess the characteristic of a component, mainly the ability to be composed at the design level. This necessitates the definition of pattern interfaces and the essential pattern properties to enable a pattern to become a design component. POAD defines a particular type of patterns called constructional design patterns and defines various levels of abstraction and logical views in designing with constructional patterns. POAD also defines how these levels are traceable to lower design levels in terms of classes.

2. **A design methodology.** We discuss a methodology to construct pattern-oriented designs. POAD is an explicit effort to study patterns as the core building blocks of object-oriented (OO) designs. In POAD, we learn from the experiences of OO analysis and design methods and define a new pattern-oriented method that builds upon the Unified Modeling Language (UML) syntax and semantics. We call the designs developed using this methodology: *pattern-oriented designs*.

3. **Real-world applications.** To study the applicability of the POAD technique, we apply the design methodology to four applications and construct a pattern-oriented design for each.

The main objective of POAD is to provide a solution to improve the practice of systematically deploying design patterns in application development. We believe that the problem should be tackled at an early phase of the development lifecycle, namely at the analysis and design levels. The approach we take is to develop the POAD methodology for building OO designs using design patterns as their building blocks.

In a software engineering context, a design methodology has three main dimensions: the *technology*, the *process*, and the *organization* aspects.

**Technology aspects.** This dimension defines the fundamentals of the design methodology, which includes the concepts, notations, and visual and formal models. We define visual models that are used to structurally glue patterns to develop pattern-oriented designs. For this purpose, we introduce the concept of pattern interfaces and discuss the relation to software components and architecture. We discuss the syntax and semantics support of UML for the POAD methodology.

**Process aspects.** This dimension defines the tasks and steps essential to develop pattern-oriented designs. Based on the visual models, analysis and design steps are defined. The output and deliverables of each step are also defined. Tool support for POAD is also discussed in the book.

**Organizational aspects** This dimension defines how the enterprise is organized to put the methodology in effect.

*Part II* of the book addresses the technology aspects. In *Part III* we discuss the process aspects. Organizational aspects are not yet addressed in the POAD methodology.
What Is Covered in this Book?

This book is mainly about the POAD methodology: its models and processes. It is also about the application of the POAD methodology to several case studies. We summarize what is covered in this book as follows:

- A Pattern-Oriented Analysis and Design (POAD) software development approach that is based on structural composition of design patterns.
- The concept of pattern-oriented frameworks and pattern-oriented designs.
- Application of the notion of design components and interfaces to design patterns and introduction of the concept of constructional design patterns.
- Application of the UML extension mechanisms to model composition of design patterns. These extensions facilitate the integration of the POAD approach to existing tools that support UML models.
- Application of the POAD design methodology to four case studies to develop pattern-oriented designs and pattern-oriented frameworks.
Chapter 2. Design Patterns and Software Engineering

This chapter provides a brief background for the topics discussed in this book. It is not intended to provide a detailed discussion on object-oriented (OO) technologies; we refer readers unfamiliar with OO concepts to one of the many traditional OO books [e.g., Booch 1994]. Understanding OO concepts and models is an essential prerequisite to using OO design patterns in creating application designs. In this chapter we mainly focus on the role design patterns play in the software engineering paradigm.
Design Patterns in the Software Lifecycle

Many application designers are motivated to utilize reusable components to reduce the effort and time of software development. The level of reusability is determined by the decisions taken on behalf of the application developer. A simple design decision, like the reuse of a library class or an API (application programming interface), is the most common and easy to use. A more sophisticated level of reuse is reusing patterns and frameworks in which the designer already takes a wider range of design decisions.

The level of reuse differs according to the reused component granularity. A standard class library provides a collection of frequently used classes and a description of how these classes can be reused together. Design patterns leverage the reuse level to the level of collaborating classes, which are sometimes called micro-architectures [Gamma et al. 1995]. OO frameworks offer higher levels of reuse in the form of a larger number of collaborating classes [Johnson & Foote 1988; Pree 1996]. Architectures constitute reusable ideas about how to structure the overall application [Shaw & Garlan 1996]. Reusable applications are the highest level in which the whole application is used. In general, reusable artifacts can be categorized according to granularity as applications, architectures, OO frameworks, design patterns, class libraries, classes, and routines.

Any software development lifecycle has several development phases, which usually include analysis, design, detailed design, implementation, and testing. Patterns can play an important role in each and every phase in the software lifecycle. Though patterns were initially introduced to serve reuse at the design phase (design patterns), many practitioners have also realized their usefulness at all other phases of the software lifecycle.

Design patterns are not another hype that will fade away; they are here to last. The more designers and developers use patterns, the more they are convinced that they should be an integrated part of any software development process. The diversity of applications in which patterns are used and have proven useful is concrete evidence of the persistency of design patterns. We quote from the second volume of Pattern-Oriented Software Architecture (POSA2) by Schmidt, Stal, Rohnert, and Buschmann [Schmidt et al. 2000]:

…the next generation of object-oriented applications and frameworks will embody patterns explicitly. Patterns will also continue to be used to document the form and content of frameworks [Schmidt et al. 1996]. Other key topics and domains that will benefit from concrete pattern mining include the following:

- **Distributed objects.** Many patterns associated with middleware and applications for concurrent and networked objects have been documented during the past decade [Schmidt et al. 1995; Lea 2000].

- **Real-time and embedded systems.** An increasing number of computing systems are embedded, including automotive control systems and car-based applications, control software for factory automation equipment [Buschmann et al. 1998], avionics mission computing [Harrison et al. 1997], and handheld computing devices. Many of these systems are subject to stringent computing resource limitations, particularly memory footprint and time constraints. Developing high-quality real-time and embedded systems is hard and remains somewhat of a "black art." Relatively few patterns have been published in this area as a result [Lange 1998].

With this obvious growing trend in discovering design patterns in many application domains and the increasing popularity of applying patterns in software development, software development methodologies that capitalize on the use of patterns in software development are essential.
POAD and OO Technology

Over the last two decades, several object-oriented analysis and design (OOAD) methodologies have been developed. Those OOAD methodologies differ in one of several dimensions. They differ in the way they approach the domain space and create the analysis and design models. They also differ in the type of models that are created to capture the output from the analysis and design process. Among those techniques, we mention the Shlaer and Mellor (1988) method, Coad and Yourdon (1990) method, Wirfs-Brock, Wilkerson, and Wiener (1990) method, Jacobson, Booch, and Rumbaugh (1992) method, Booch (1994) method, the Objectory [Objectory 1998] method, the Hierarchical Object-oriented Design (HOOD) [Robinson 1992; Hood 1993] method, the Real-Time Object-oriented Modeling language method [ROOM 1994], the Catalysis [D'Souza & Wills 1998] method, and much more.

The variety of OOAD methods prompted the development of Unified Modeling Language (UML) as the language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. Prior to UML, several modeling languages have been available, most of which share a set of commonly accepted concepts that are expressed differently. This lack of agreement discourages users from using OO technology.

UML is not only a descriptive visual language, but it has both syntax and semantics capabilities as well. UML notation represents the graphical syntax for expressing the semantics described by the underlying UML metamodel. The UML metamodel provides a common and definitive statement of the syntax and semantics of the UML elements. For further details on UML syntax and semantics, see www.uml.org. In summary, the models supported by UML are

- **Use case diagrams.** A use case defines a situation in which the system is used, its inputs, and possible outcomes. Use cases are further analyzed to produce possible scenarios. The collection of scenarios is one aspect of the requirement specifications of the application. Use case diagrams are models used by the analyst to represent all possible usages of the application (use cases). The diagram also shows the relationships between the use cases and hence establishes relationships between the various application functionalities. Use case diagrams also illustrate how external entities interact with the application, whether these entities are other external components, applications, or users.

- **Class diagrams.** A class diagram describes one structure aspect of the application. It is a representation of possible classes and their relationships. Detailed class diagrams reveal details about the design, which include the attributes and their types, methods and their signatures. Class diagrams are the essential structure models to any OOAD methodology, since all OO programming languages provide direct support for elements captured in class diagrams.

- **Behavior diagrams.** These diagrams explain behavioral aspects of the applications.
  - **Statechart diagram.** Statecharts are techniques to model the behavior of large, complex systems. UML statecharts are based on David Harel’s statecharts [Harel 1988]. The behavior of objects in an OO design can be studied using statecharts support in UML.

- **Interaction diagrams.** Interaction diagrams are either sequence or collaboration diagrams:
  - **Sequence diagrams.** Object scenario diagrams are means to record design decisions by examining the interaction between objects in the application in a timely, ordered manner. Normally, there are one or more scenarios for each use case.
  - **Collaboration diagrams.** Collaboration diagrams are another view of scenarios without the timely sequence ordering. Their objective is to show how objects interact and send messages to each other. Concurrent threads of execution can also be represented.

- **Implementation diagrams.** Implementation diagrams capture the runtime view of the application. Whereas UML diagrams capture the analysis and design time views of the various modules in the application, implementation diagrams capture the deployment and runtime aspects.

- **Component diagram.** A component diagram shows the application components and how they interact. It describes a static view of the system. **Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations [Booch et al. 1999].** Components are used to model the executable pieces of the
application and their relationships as expected at runtime.

- **Deployment diagram.** The deployment diagram shows the system's physical architecture in terms of nodes and their relationships. Deployment diagrams are related to component diagrams in that a node typically encompasses one or more components. Means are provided to identify which components will be executing on which machines.

These diagrams provide multiple viewpoints of the system under development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. Models used for OO analysis and design can be generally classified as static and dynamic models. Static models describe the structure of the application, the high level functional and data organization, as well as the broad information content. Static views include class diagrams, package diagrams, component diagrams, and deployment diagrams. Dynamic views include use cases, scenario and collaboration diagrams, and the behavioral aspects of individual object using statecharts.

UML provides several modeling constructs that can be used to model a pattern as a first class design element. It also provides models to capture the internal structure and behavior aspect of a pattern. Chapter 6 discusses in detail UML support for modeling design patterns. In POAD we use UML as a means to capture all the models generated from the development process.
Design Patterns

In this section we explain what a design pattern means and briefly discuss design patterns evolution. We also present a lifecycle model for patterns showing the various development and usage phases. This lifecycle helps in establishing trust in patterns as high-quality designs.

What Is a Pattern?

In general, a pattern describes a problem that frequently occurs in software design and implementation, and then describes the solution to that problem in such a way that it can be reused. Patterns are introduced to document good design practices; they are the vehicles of knowledge and experience transfer from experts to the novice. For this purpose, much work has been motivated to document and discover new patterns in various domains. Patterns can be classified, according to the development phase in which they are used, into analysis patterns [Fowler 1997], architecture patterns [Buschmann et al. 1996], design patterns [Gamma et al. 1995], and idioms [Coplien 1992]:

- **Analysis Patterns.** Analysis involves looking behind the surface of requirements to understand the problem. Martin Fowler [Fowler 1997] defined analysis patterns as “…groups of concepts that represent a common construction in business modeling.” Fowler documented several analysis patterns out of practical project experiences for several business domains. The Type, Observation, and Measurement patterns are among those documented by Fowler.

- **Architecture Patterns.** An architecture pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems or components, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [Buschmann et al. 1996]. The Broker, Blackboard, and Filters and Pipes are among patterns in this category. Architectural patterns are a means of documenting architecture for complex and heterogeneous systems, thus helping to manage the application complexity.

- **Design Patterns.** A design pattern provides a schema for refining the subsystems or components of a software system or the relationship between them [Buschmann et al. 1996]. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context [Gamma et al. 1995]. The Strategy, State, and Proxy patterns are examples from this category.

- **Idioms.** An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of a given programming language such as C++, Java, or Smalltalk. Examples of idioms are how to implement Singletons in C++ [Buschmann et al. 1996] and the Counted Pointer [Coplien 1992].

Patterns are well-proven design experiences. It is often said that patterns are discovered or documented rather than invented because they have to evolve from more than one existing practical project. Patterns offer concise and efficient ways to convey software concepts and experiences in real problems.

History of Patterns

The idea of software patterns evolved from various initiatives. The architect Christopher Alexander, a professor of architecture at the University of California at Berkeley, developed the foundation for patterns. The word pattern has been related almost entirely to his work. He and his research group spent over 20 years developing an approach to civil architectures using patterns. Alexander described over 250
patterns over a wide range of abstraction, from town structures to room designs. He founded the fundamental descriptive template of a pattern as a context-problem-solution. He developed a prototype of pattern books from his work in pattern cataloging for architectures [Alexander et al. 1977; Alexander 1979].

Kent Beck and Ward Cunningham got enthusiastic about applying Alexander's ideas to software development. They wrote the first set of patterns, which were specific to user interfaces. Kent focused on idioms for Smalltalk, and Ward captured his experience with business systems (accounting applications).

The first published work about the use of patterns in software development was Erich Gamma's 1991 doctoral thesis [Gamma 1991]. Written in German, this work did not receive much publicity. "Nearly half of the patterns described later [in Gamma et al. 1995] were previously documented in his thesis" [Buschmann et al. 1996]. Bruce Anderson is one of the leaders in the pattern community; he had a workshop on patterns at OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications conference) in the early 1990s. Jim Coplien's described idioms in C++ in his book Advanced C++ Programming Styles and Idioms. Those idioms were somehow related to the idea of documenting solutions to frequent problems. A group called the Hillside Group was formed to further explore these ideas and promote the use of patterns in software development. They worked to lead and support new members in the pattern community. This group formed the first PLoP (Pattern Languages of Programs) conference in 1994.

The great public knowledge of the pattern movement was triggered by the Gang of Four (GoF) book, Design Patterns: Elements of Object-Oriented Software [Gamma et al. 1995], which presented the first well-described and documented catalog of design patterns for OO designs. Gamma, Helm, Johnson, and Vlissides' work represented a classification of commonly used and well-known design solutions in the OO paradigm. They documented a set of twenty-three patterns under three categories: Behavioral, Structural, and Creational.

Peter Coad had an early work on OO patterns [Coad 1992] in which he described seven simple patterns in OO analysis and design. He worked on patterns for analyzing a given application domain and using OO technology to build applications [Coad 1995]. Douglas Schmidt is another leading pioneer in the pattern community; he is the author of many patterns in communications systems and distributed applications [Schmidt 1996]. Wolfgang Pree has worked on patterns for framework development [Pree 1994, 1995, 1996]. He categorized the structural principles into metapatterns used to develop frameworks and addressed the Hot-Spot and the Hooks/Templates approach in framework development.

Pattern-Oriented Software Architecture: A Pattern System, also called the "Gang of Five" book [Buschmann et al. 1996], addressed the use of patterns at the architectural level of software development. The authors classified the software patterns as architectural patterns, design patterns, and idioms. Most of their contribution was geared toward architectural patterns. Their book, together with the GoF book, represents the starting point for novices in the pattern community. Volume 2 of the POSA series [Schmidt et al. 2000], entitled Patterns for Concurrent and Networked Objects, documents patterns and past practices that represent well-established techniques for building distributed applications. The book includes service access and configuration patterns, event handling patterns, synchronization patterns, and concurrency patterns. A pattern language for middleware and applications is also presented.

Several conferences have been held to evaluate and document new patterns. The yearly Pattern Language of Programs conferences (PLoP and EuroPLoP) are the main sources of pattern evaluation and cataloging. Further matured patterns are documented in a series of books [PLoPD, PLoPD2, PLoPD3, PLoPD4].

The Pattern Lifecycle

Figure 2-1 illustrates the lifecycle of a pattern. We use this lifecycle model to establish trust in those patterns chosen as design components by showing that several improvement iterations are exercised on the pattern before it is documented for reuse.

**Phase 1: Mining.** In the creation of any design pattern, the first phase is concerned with the initial documentation of a pattern. The primary activity in this phase is mining for patterns. Experienced authors, working in practical projects, discover and document patterns. Authors can also mine for patterns in their projects [Buschmann et al. 1996]. The author of a pattern has to decide what constitutes a pattern, what makes it reusable by others, what domain it serves, and hence whether or not it qualifies to be a pattern. Some practitioners in the pattern community view the "rule of three" sufficient to document a pattern (i.e., usage of the pattern in three or more practical projects). The output of this phase is a version of the pattern as documented by the author.

**Phase 2: Polishing.** The second phase is concerned with evaluating and improving the pattern by experienced practitioners and researchers. In this phase the pattern author submits the pattern to one of the yearly PLoP conferences. The submission is then assigned to a reviewer. In the pattern community a reviewer is actually a
shepherd who works with the author on a peer-to-peer review process to improve the pattern. At the end of the review process, the pattern is either admitted to the conference or rejected. The accepted patterns are then reviewed again during the conference with a group of experienced authors who make suggestions for improvements to the pattern and share their experiences in solving similar problems. The objective of this phase is to help the pattern author improve his or her version of the pattern and in some cases abandon the pattern because of lack of soundness and potential for reuse. The output of this phase is a well-documented pattern for usage by novices, application designers, and developers. The revised version is then published in the conference proceedings. For details about this phase, refer to the documentation of the writers’ workshop that takes place in these PLoP conferences [Gabriel 2002].

Phase 3: Reuse. The third phase is concerned with the actual reuse of the pattern in applications. Pattern users look for patterns in one of the published PLoP conference proceedings or PLoPD (Pattern Language Of Program Design) books. They then instantiate the pattern in practical projects. Users provide feedback to the pattern author on the obstacles they faced during implementation and use, and make suggestions for improvements.

Figure 2-1. The pattern lifecycle.

As shown in Figure 2-1, the process is iterative, and hence a pattern is continuously improved. We note that the pattern documented for reuse is expected to be of high quality, since it has gone through several phases of improvements. High quality is an essential attribute of a design component, and hence patterns following this development lifecycle qualify to be design components.
Design Frameworks

In this section we discuss the concept of design frameworks and the role of design patterns in developing frameworks.

What Is a Framework?

The term framework has different definitions according to the context in which it is mentioned. Generally, a framework is a frame on which (or around which) something is made (built). Fayad and Schmidt (1997) present a comprehensive discussion on OO frameworks, in which they classify application frameworks and discuss their strengths and weaknesses in a broad sense. Johnson and Foot (1988) describe a framework as a reusable, semicomplete application that can be specialized to produce custom applications. In that context, a framework classification is introduced as white box and black box frameworks. For black box frameworks, the source code of the original framework cannot be modified but can only be extended. White box frameworks require understanding of the framework’s structure and the hot spots to which application-specific functions will be hooked. Another classification of frameworks as application or domain-specific frameworks was introduced in Hans Schmid's 1996 article "Creating Applications from Components: A Manufacturing Framework Design." Application-specific frameworks provide the basic functionalities of a working application, but the specific contents that model the application domain have to be added for each particular application. Domain-specific frameworks are less commonly used. They model the domain-specific functionality using common objects and generic application logic that pertain to a particular domain. An application is built by configuring these objects and binding the generic application logic to the configuration.

Other views of frameworks include frameworks as "a class library that captures patterns of interaction between objects and consists of a suite of concrete and abstract classes explicitly designed to be used together. Applications are developed from a framework by completion and implementation of abstract classes" [Rogers 1997] and as "a partially complete software system that is intended to be instantiated. It defines the architecture for a family of systems and provides the basic building blocks to create them. It also specifies the places where adaptations for specific functionality should be made. In an object-oriented environment a framework consists of abstract and concrete classes" [Buschmann et al. 1996].

As we discern from the above discussion, the term software framework is very generic and must be qualified by attributes from the context in which it is used, such as domain-specific framework, white-box framework, design framework, and so on. In the following subsection, we focus on design frameworks, which are a collection of design classes that are generic for a particular domain and provide the necessary mechanisms for extension and customization at design time.

Developing Robust Designs using Frameworks

Design frameworks can greatly facilitate the design and implementation of large-scale software systems. They enable reuse at higher levels of abstraction than do component or code reuse. A framework provides a set of services for the application domain and should work within the execution environment of that domain. Domain-specific frameworks result in a robust design that is maintainable and supports a rapid development lifecycle.

Examples of frameworks include Java Foundation Classes (JFC), Microsoft Foundation classes (MFC), and Borland's Object Windows Library (OWL). These frameworks have greatly facilitated the rapid development of robust designs. Complex windows applications based on object linking and embedding (OLE), database services, multiple document windows, and network services can be developed in an incremental process supported by a software architecture based on these frameworks.

A frequently used example for frameworks at the implementation level is a GUI framework such as Java's AWT and Swing. These frameworks provide classes and interfaces for GUI functions. Specialized widgets can be developed by subclassing from the Swing
classes and overriding certain methods.

Other examples of frameworks for distributed applications are the Common Object Request Broker architecture (CORBA), Distributed Component Object Model (DCOM), and enterprise Java beans (EJB). These middle-tier frameworks allow the development of software based on a technology that enables interoperability among disparate systems running on heterogeneous platforms.

Frameworks for real-time applications can also provide great benefits in rapid development of software architectures that satisfy rigid timing and safety requirements and environmental constraints [Douglass 1999].

The Role of Patterns in Developing OO Frameworks

The definitions of framework show that many authors have different interpretations of what constitutes a framework. These definitions are not conflicting; they describe several aspects of frameworks, such as the structure of a framework (how it is constructed), the instantiation of a framework (how it is used), and classification of frameworks (black box, white box, application-specific, domain-specific). Most of the work refers to frameworks as collaborating classes, and whenever patterns are used, they are presented as class participants. Beck and Johnson, in their work “Patterns Generate Architectures” show how patterns can be used to derive the HotDraw architecture in which patterns are used to facilitate the understanding of the final system. In this book, we refer to pattern-oriented frameworks as design frameworks defined by an architecture of communicating design patterns (in a pattern-level view), which are further extended to classes and objects (in a class-level view).

Patterns can be thought of as micro-architecture elements, and hence a single framework contains many patterns [Johnson 1997]. This viewpoint closely encourages framework designers to shift the view from collaborating classes to collaborating design patterns. Conversely, a refactoring approach can be used to create generic designs from existing application designs and then reuse the generic design in creating new applications within the same problem domain [Castellani & Liao 1998]. This approach starts with an existing application, then abstracts design macrocomponents through the removal of application specifics. In that context, patterns can be useful in the refactoring process.

Patterns can also play a good role in documenting designs. They can be used as an intermediate level of systems description between the analysis and design levels [Odenthal & Quibeldey-Cirkel 1997]. The pattern level reduces the descriptive complexity by covering the design with pattern instances. Identifying candidate patterns is closely related to the pattern-oriented framework construction.

As an example of using patterns in domain-specific frameworks, Hans Schmid presents an OO framework for manufacturing systems. He starts from an existing OO analysis and targets a more general and flexible architecture for automated manufacturing systems like an assembly line. Although successive transformation steps are guided by design patterns to create an overall architecture, patterns are not used as design building blocks [Schmid 1995].

It is clear from these experiences that patterns play an important role in the design of frameworks, since they provide the necessary abstraction that can be used to distinguish domain-specific (hot spots) and generic parts (fixed spots) of the framework. Moreover, the use of patterns in developing design frameworks improves the understandability and documentation of the framework.
Design Pattern Composition Approaches

Several successful experiences have reported on the advantages of using patterns in designing applications. For instance, Sirinivasan and Vergo, in "Object-Oriented Reuse: Experience in Developing a Framework for Speech Recognition Applications," discuss their experience in using design patterns to develop highly interactive software systems for speech recognition. Garlow, Holmes, and Mowbary, in "Applying Design Patterns in UML," describe another experience in applying analysis, architecture, and design patterns to design and implement OO cellular communication software. These experiences, among others, do not follow a systematic method to develop applications using patterns. Systematic development using patterns utilizes a composition mechanism to glue patterns together at the design level. Generally, we categorize composition mechanisms as behavioral and structural compositions. These approaches are discussed in detail in the following chapter.
Summary

Almost all OOAD methodologies focus on identifying classes and objects and providing the necessary models to perform the discovery and modeling process. With the new shift in software development towards using patterns, we realize that those OOAD methodologies do not provide explicit support for identifying patterns and composing them. We seek techniques that use patterns as first-class design artifacts that should be treated with the same level of importance that we give to classes and objects in traditional OO design methodologies. In the following chapter we summarize some of the techniques that shift the focus from traditional objects and classes to patterns as units for composition.
Chapter 3. Composition of Design Patterns

The development of applications using design patterns as design components requires a careful look at composition techniques. In this chapter we briefly review several techniques for composing design patterns. These techniques can be categorized as

- Behavioral composition techniques
- Structural composition techniques

The behavioral techniques are based on object interaction specifications to show how instantiations of patterns can be composed, whereas structural techniques are based on the static architectural specifications of composed instantiated patterns using class diagrams. Although we adopt in the following chapters the structural approach, a hybrid technique showing both structural composition and behavioral composition may evolve as the more comprehensive approach for specifying how instantiated patterns can be composed.

Analysis of pattern behaviors is an important topic, especially when those behaviors are composed together to develop an application design. Behavior composition could result in an unreliable design in which design faults are introduced due to unanticipated interactions between individual design components. These interactions must be analyzed using the behavior specifications of these components. It is important to note here that structural composition techniques could suffer even more from behavioral inconsistencies; hence, some designers prefer behavioral composition techniques. Whether a structural or a behavioral composition approach is used, careful behavioral analysis of the composed design is needed in order to detect such interaction violations. It is also important to note that several architectural design patterns are intended to produce designs in which interactions between objects are controlled by specific methods that tend to minimize coupling between objects and hence provide designs with controlled object interactions. The methodology that we present in this book is based on defining an interface for each design pattern instantiated. This interface provides a specification for the interactions between objects in different instantiated patterns and facilitates behavioral analysis.

The increasing numbers of patterns and pattern languages have prompted the development of classification techniques for patterns and pattern relationships. This goes back to the first book on patterns in which classifications of creational, structural, and behavioral patterns were introduced. A pattern map was also developed showing relationships between the 23 patterns described in Design Patterns: Elements of Object-Oriented Software [Gamma et al. 1995]. Since the text of each pattern could describe its relationship to other patterns, the pattern literature showed different classification schema of these relationships. James Noble developed a classification schema for the relationships between patterns [Noble 1998]. Three primary relationships and a number of secondary relationships between patterns are identified. The primary relationships are based on the following:

- Uses relation, where a pattern "uses" another pattern
- Refines relation, where a pattern "refines" another pattern
- Conflicts relation, where a pattern "conflicts" with another pattern

Typically, more complex patterns "use" simpler patterns. For example, the Model-View-Controller (MVC) pattern [Buschmann et al. 1996] uses instances of the Observer pattern, the Strategy pattern, and the Composite pattern. The "refines" relationship can be used to define a new pattern as a refinement of a published pattern. The "confines" relationship shows that instances of these patterns should not be used together—that is, they provide mutually exclusive solutions to the same problem, and it is not possible to use them interchangeably. Noble also discusses a number of secondary relationships and shows how they could be defined using the primary relationships. This classification schema helps in the process of selecting patterns from pattern databases during the analysis phase of POAD, as we discuss later in Part III.

The main problem we discuss in this chapter is not how patterns relate to each other but rather how pattern instances can be composed together as building blocks to develop composite patterns, OO applications, or OO frameworks. This is related to the problem of composing software components at the design level. Understanding the relationships between individual patterns is a good practice but does not solve the issues related to pattern composition.
Behavioral Composition Techniques

Behavioral composition approaches are concerned with objects as elements that play multiple roles, where each role is part of a separate pattern. These approaches are also known in the OO literature as interaction-oriented or responsibility-driven composition [Wirfs-Brock & Wilkerson 1989]. Although the POAD composition approach, which is the main topic of this book, uses notation and composition techniques that are based on the pattern structure (i.e., its class model), we find it useful to be familiar with existing composition techniques that utilize the pattern’s behavior model.

In this section we summarize behavioral approaches in modeling and composing patterns, and we analyze their advantages and drawbacks. Formalizing the behavior specification of individual patterns is important for the purpose of clarifying their semantics and facilitating their utilization by any pattern composition approach. We briefly summarize the work in the behavior specification field. We then discuss the approach presented by Trygve Reenskaug [Reenskaug 1996] on role modeling and synthesis using the OO role analysis method. We describe the work by Dirk Riehle [Riehle 1997] presented at the OOPSLA conference in 1997. This work applies the concepts of role models suggested by Reenskaug to pattern composition. Then, we briefly summarize the superimposition approach by Jan Bosch [Bosch 1998c], which uses design patterns and frameworks as architectural fragments and merges roles and components to produce applications. Finally, we discuss the role/type/class three-layer approach developed by Lauder and Kent [Lauder & Kent 1998], which takes a visual specification approach to describe design patterns.

Object-Oriented Role Analysis and Software Synthesis

The concepts of role models have emerged as a software analysis and design technique. Trygve Reenskaug developed the Object-Oriented Role Analysis and Software Synthesis (OORASS, later called OOram) [Reenskaug 1992; Reenskaug 1996]. Reenskaug presents a role model that abstracts the traditional object model and additionally recognizes a pattern of objects and describes it using a corresponding pattern of roles. While the notion of classes focuses on the capabilities of objects that are instantiated from these classes, the notion of roles focuses on the responsibilities of an object within the overall group of objects. According to Reenskaug, “A role is an architectural representation of the objects occupying the corresponding positions in the object system.”

The OOram method specifically addresses two development processes: the modeling process during which role models are created and the synthesis process during which several role models are composed together. In the following we will give an overview of the two processes.

A role model is a collaboration of objects that the analyst chooses to regard as a unit, separated from the rest of the application during some period of consideration or during analysis. In developing the system, the designer would then consider developing the whole application as a collection of role models, or so-called role synthesis [Anderson & Reenskaug 1992]. In role modeling, we suppress irrelevant objects, aspects, and details and generalize object identity. In role synthesis, new role models are derived such that every role is the composition of one or more roles from the base role models. The derived role combines several roles to be treated as a new single (composite) role.

During the modeling process, the real world is analyzed to identify a number of objects and their collaborations. This process is similar to many OO analysis methods in which we use requirements analysis techniques to identify objects, their types, and how they interact with each other. Based on objects’ interactions, the role played by each object is identified. A role model is then created to separate different concerns. A role model describes the subject of a specific interaction, the relationship between objects (playing specific roles in the interaction), and the messages exchanged by objects. A single object can play multiple roles; each role played by the object belongs to a specific collaboration or role model. For instance, assume that we are building the enterprise information system for an organization. An employee in this organization may play multiple roles. He could be the traveler in the travel account model or a project manager in the payroll model. Each role model describes a limited aspect of the problem. Hence, when we do role modeling, we end up with several role models. The object plays different roles in those models.

In OOram roles are used as an abstraction to the traditional notion of classes and objects. Like a class, the role is a description of a set of objects. Whereas a class describes a set of objects with common behavior and characteristics, a role describes a synergy of several (not
necessarily common) behaviors and characteristics. In addition, roles have properties similar to objects. In a role model, we can define the messages exchanged between roles and how they react to those messages.

The rationale behind using role models to capture both the static (class) and the dynamic (object) aspects of the model is that during modeling, the modeler would like to see the capabilities of the model elements (which are traditionally captured in class diagrams) as well as the behavior of the model elements and their message exchange (which are traditionally captured in object diagrams). OOram creators believe that while class diagrams are good at the implementation and development phase, they are not appropriate for the modeling phase, since they only capture the "type" information, while most message collaborations should be captured with those types as well. In the OOram modeling process, the analyst starts by analyzing the real-world system and identifying objects and their interactions. Then, the analyst abstracts those object interactions into role models. Figure 3-1 illustrates the modeling/abstraction process.

Figure 3-1. The OOram role modeling process.

The abstraction process considers focusing on individual collaborations at a time and discarding other collaborations in the object model. For instance, the analyst may drop objects of no interest (i.e., not relevant to a particular collaboration), suppress irrelevant aspects or details, or generalize the objects used in the collaboration. The result is a set of role models that are abstractions from the object diagrams. Objects in that sense become instances from the role and object collaborations become instances from role models.

The role model diagrams have constructs for modeling: a system role and its attributes, an environment role, a message path between collaborating roles, ports through which messages can be delivered to other roles, means to explicitly model a role's knowledge of another collaborating role, and means to model role and port multiplicity. An example of the role model view is illustrated in Figure 3-2. The example is taken from an airline booking system where several roles collaborate to make booking and payment.

Figure 3-2. An example of a role model for airline booking [Reenskaug 1996].

In addition to the role model views, OOram provides other views: a scenario view, which is similar to UML interaction or scenario diagram; an interface view, which is an enhanced role model that illustrates the exact messages over each collaboration arc; and method specification view, which is a scenario view with more details about the message processing inside the role.

The second process in the OOram method is the synthesis process. The synthesis process integrates various model views into one synthesized model view. The synthesis of base role models produces a synthesized (derived) role model. The synthesized role model consists of new roles and their collaborations. A composite role in the synthesized role model would then play various simple roles from other base role models.

The synthesis process combines role models (collaboration views), interface views, scenarios views, and method specification views. For the role model view, OOram specifies several rules for the synthesis process:

1. All roles from the base models should be mapped to roles in the final model.
2. Role attributes and collaborations should be mapped to attributes and collaborations in the final role model.
3. All ports in the base models should be mapped into corresponding ports in the final synthesized models.

The views capturing the synthesis process are complex even with few numbers of roles in each role model. Figure 3-3 illustrates the synthesis of two role models: one is the airline booking role model (top portion of the figure) and the other from the travel expense role model (bottom portion of the figure).

**Figure 3-3. OOram synthesis of role models [Reenskaug 1996].**

OOram finds role models appropriate for modeling design patterns where the pattern solution involves collaboration of interacting objects. A role base model could be created to represent the abstraction provided by the pattern. This pattern role model can be the source of synthesis. When patterns are modeled using a role model, the OOram synthesis procedure can then be used to compose patterns together. Composition of patterns using role models is not covered exhaustively as part of the OOram method. Dirk Riehle used role models to compose simple patterns and create composite design patterns, as discussed in the next section [Riehle 1997].
Role models used by Reenskaug for modeling design patterns differ from the archetypal object structures defined as design patterns in the pattern community. Reenskaug uses one model for both static and dynamic aspects of a role. Modeling a design pattern using roles includes both the static and dynamic aspects of the pattern in one diagram. The Unified Modeling Language (UML) avoids such integration. Static models are used for composition and interfaces purposes, while dynamic models are used for behavioral analysis and interaction purposes. Therefore, it is always better to keep separate models for each; however, maintain their relationship and consistencies.

In OOram, traceability of roles assigned to objects is a visual problem in the model synthesis. The diagrams of synthesized role models can be overwhelming with directed arrows synthesizing the roles from several role models. To overcome this problem, a tabular presentation of role models is suggested. Tabular representation is much less illustrative than visual models, such as collaborations, scenarios, or role models. Reenskaug shows how to document a framework using patterns described as role models (the MVC) similar to that described by Ralph Johnson in "Documenting Frameworks Using Patterns" (1992) with class diagrams.

Other work in behavior composition using roles can be found in Gottlob, Schrefl, and Rock's "Extending Object-Oriented Systems with Roles" (1996) and Kristensen and Osterbye's "Roles: Conceptual Abstraction Theory and Practice Language Issues" (1996). Kristensen and Osterbye focused on using roles in the context of conceptual modeling. In this context, roles are treated as first-class elements in the analysis of the application. A role represents a perspective on some objects as it emerges. Objects can be seen as role instances. Roles can be aggregated, specialized, and generalized. Their approach suffers from some limitations [Riehle 1996]. For example, how are roles attached to objects? How is the state of a role managed? How are state transitions of roles handled if they cross a specific role boundary?

Composing Design Patterns Using Roles

Role diagrams are developed to document the behavior of design patterns. Riehle (1997) uses role diagrams for composing design patterns. The role-based approach considers a pattern as a collaboration of roles. Objects and roles are treated slightly different. A role is a specific behavior defined as a result of collaboration and interaction with other roles. A design pattern captures a behavior that can be modeled using a set of interacting roles. Objects are usually used to refer to application specific objects. An object will play multiple roles in which each role defines the object behavior in a particular communication with other objects. To compose patterns, each pattern will be represented as a role diagram (a set or roles with some predefined relationships). The role diagrams are then composed to objects to develop an object diagram for the application.

Riehle investigated a particular composition problem related to what he calls composite design patterns. Composite design patterns are different from the GoF Composite pattern; whereas the Composite pattern is a design to capture hierarchy in OO applications, composite design patterns are composition of patterns whose integration shows a synergy that makes the composition more than just the sum of its parts. Thus a composite design pattern is simply a set of design patterns integrated and composed together so that the integration satisfies the condition of being a pattern; that is, it solves a recurring design problem.

To be more specific, Riehle discusses the process to create composite design patterns and does not address the general problem of composing any patterns. Though the problem is different, we can learn from the synergy and synthesis techniques used in developing composite design patterns. To create composite patterns, Riehle follows a simple procedure:

1. We model all existing patterns using role diagrams. Role diagrams are extensions to role models [Reenskaug 1996]. In addition, a set of composition constraints is defined in role diagrams. These constraints include the following:
   - An object playing a role (A) always plays another role (B) in the same collaboration, thus role A implies role B;
   - An object playing role (A) never plays role (B); and
   - Arbitrary mix two roles.

   Role diagrams for several patterns, including Mediator, Observer, Chain of Responsibility, and Composite patterns, are illustrated in Dirk Riehle's "Composite Design Patterns" (1997).  

2. The design uses a prototypical pattern application to derive the composite design pattern. A prototypical application is a concrete application, which is usually represented by an object diagram. It plays the role of the concrete example to be abstracted.
3. Using the collaboration diagram for the application, the designer starts to select the patterns that can be used in that application. He or she uses the role diagram of each pattern and assigns roles from the role diagram to objects in the application object diagram. At the end of this process, each object will be assigned several roles from several patterns.

4. Using the object diagram with the annotated roles, the designer creates a role relationship matrix. The role relationship matrix is used to analyze how roles are close to each other in terms of the composition constraints. The purpose of this analysis is to discover the pattern interaction synergy and unleash any composite roles. As a result, role equivalent sets that form composite roles are defined. The role relationship matrix is reduced to a final role relationship matrix.

5. The final role relationship matrix is used to create the role diagram for the composite design pattern.

In order to use this approach as a composition technique for design patterns in general, more work needs to be done in the way roles are assigned to objects. Moreover, we need to address the issue that we do not have domain objects (or application classes) to which we assign roles; instead, we want the role diagrams of individual patterns to derive the application-specific classes and objects.

The main drawbacks in modeling composite design patterns as compositions of role diagrams are identified as follows:

- "Choosing role diagrams as the primary means for describing pattern ignores class inheritance based patterns for which further techniques have to be developed" [Riehle 1997]. The POAD approach works with patterns as structures of related classes and thus addresses the inheritance problem in role modeling.

- Some patterns tend to be more complex when represented by role diagrams, specifically patterns with recursive structure like Composite or Chain of Responsibility patterns because "...satisfying the boundary conditions increases the number of roles and composition constraints" [Riehle 1997].

- The process of composing role diagrams is an after-the-fact realization. "...The creative process of working out the pattern did not proceed in the linear fashion as implied by the steps taken" [Riehle 1997].

- Role diagrams are not explicitly adopted by UML; the Object Constraint Language (OCL) and some extensions to collaboration diagrams can be used, however. One of the main obstacles confronting the adoption of OO approaches is the notation and semantics of modeling languages. UML finally brings OO modeling approaches together and unifies OO models. Therefore, new modeling approaches should be integrative with UML models.

- There is no consensus on a unique role diagram for a design pattern. For instance, disagreement arises between the representation of Chain of Responsibility and Composite patterns [Riehle 1996; Reenskaug 1996]. Patterns are usually documented using class diagrams or object collaboration diagrams. Pattern authors do not use role models to document their patterns.

The main advantage of using a role diagram to model a pattern is that it provides a higher level of abstraction than class diagrams. A role model abstracts the main idea behind the pattern, while its class diagram provides implementation templates that could be several for one pattern (for example, several templates for the Observer pattern as compared to one role abstraction diagram [Riehle 1996]).

**Architecture Fragments and Superimposition**

The concepts of architectural fragments develop further from the concepts of role models. The approach by Jan Bosch in "Specifying Frameworks and Design Patterns as Architecture Fragments" (1998c) uses design patterns and frameworks as architectural fragments and merges roles and components to produce applications. Bosch introduces the notion of architectural fragments as a means to represent design patterns as reusable first-class entities. Fragments are used by Bosch to address the problem of underrepresentation of patterns in traditional OO languages, which provide no explicit support for representing a pattern as a first class element.

The approach is based on separation of behaviors. Each pattern captures a well-defined behavior. The behavior of a design pattern can be described as the behavior of a group of communicating objects. This behavior can be captured explicitly and regardless of the rest of the application functionality. An architecture fragment is used to capture this behavior. However, this behavior is not in isolation. Several of those behaviors need to be integrated and composed together. Therefore, a fragment composition approach is needed. Bosch introduces two techniques: first, a way to represent the behavior of a pattern (fragments), and second, a way to compose those behaviors (superimposition).

For a set of classes, each class plays a role in a specific behavior. An architectural fragment describes the parts of each of those classes...
that are specific for the specific behavior captured by the fragment. Hence, a fragment consists of a set of roles. A role is similar to a class. In addition, it defines a required interface. The required interface specifies the conditions that any class should meet if it is required to implement that role.

To compose patterns using the architecture fragment approach, each pattern is represented as a fragment. We may also have domain-specific design that contains classes (generated by the designer but not part of a particular pattern). Given a fragment representation for patterns and domain classes, a superimposition technique is used to compose those fragments together. Figure 3-4 illustrates the approach [Bosch 1998c]. This figure shows two fragments; the first has three roles and the second has two roles. It also has three domain classes. The result of the composition is a set of application classes that play several roles. For instance, an application class can play three roles: a role from fragment one, a role from fragment two, and a domain class.

Figure 3-4. Using fragments and superimposition to develop application classes.

In order to make this mechanism feasible, a specification language should be used to represent roles. A role could be as simple as a specification of interface operations or as complete as a specification for a whole class. In addition, a fragment representation has a way to specify what is required from any class that wishes to play a specific role that is part of that fragment. A layered object model (LayOM) [Bosch 1998b] language is used to specify roles and fragments.

As an example, consider the Observer pattern [Gamma et al. 1995]. This pattern as a fragment is composed of two roles: the subject and the observer. The subject role contains a set of objects, that is, observers that depend on the subject. It contains methods for subscribing and removing dependents. The observer role defines the interface to be called by the subject in case it changes. The Observer pattern representation using LayOM is illustrated in [Bosch 1998c]. One important observation that we make on the use of fragment roles to describe a pattern is that the class diagram of the pattern is not directly mapped to fragment roles. For instance, the Observer pattern class diagram has at least four classes with two abstract classes, representing the abstract subject and the abstract observer, and two or more concrete classes implementing the abstract classes and providing the application specific details. In the fragment modeling approach, the Observer pattern contains two fragment roles only.

A fragment consists of a number of roles. A role consists of variables, methods, acquaintances, layers, and interfaces as defined by the LayOM language. When designing the architecture, the designer selects a set of fragments to use and creates a set of domain-specific classes. The designer will have to choose which domain class will play the roles of those architecture fragments. The designer can also integrate fragments together by merging their roles. In order to do so, for a fragment-fragment composition, the role of one fragment has to be composed with the role of other fragments and for a fragment-component (class) composition, the role of a fragment has to be composed with the component's behavior. This composition would sometimes require overriding the component's behavior by the role to be superimposed on the component.

A superimposition mechanism is used by Bosch to compose roles and their behaviors. This superimposition mechanism is not supported by traditional languages. Bosch created a technique required for superimposition in the LayOM language. Through the use of layers, LayOM provides various techniques to compose behaviors. "Due to the notion of superimposition and the LayOM layers that provide an implementation means, an advanced composition method is provided that facilitates the composition of roles and components. They provide the basic features necessary for specifying reusable architectural fragments that can be composed in powerful ways with suitable reusable components" [Bosch 1998c].
The superimposition approach is developed to impose behavior on components using the fragment approach. It suffers from some limitation, especially when it comes to expressing OO techniques that are used in the class diagram level, such as inheritance. Since one of the important models used in documenting patterns is a class diagram, it is more likely that the fragment and superimposition approach will suffer from some limitation in implementing class model mechanism like subclassing.

This approach (as well as similar ones like aspect-oriented programming (AOP) [Kiczales et al. 1997]) suffers from a common drawback: they require the use of specific language to model the composition. For example, LayOM is required to model fragments and superimposition. In AOP, various aspects are described using a special aspect language that requires a specific compiler to weave aspects and merge the domain functionality with the aspect code. Hence, the designer has to learn an additional modeling language that is not part of UML (though some mapping and extensions could be made).

Bosch’s approach uses the superimposition mechanism to allow a class to play several roles in architecture fragments. This approach is targeted for lower level implementation of patterns and frameworks and assumes the existence of application-specific classes to which we want to assign a role (i.e., merging process). It is a behavior composition approach of roles. The POAD approach is a structural composition approach that uses pattern views that comply with UML models, which we believe could be simpler (and more familiar) for the designer to use.

Role/Type/Class Modeling

Visual modeling is one of the important aspects of modern software development. Lauder and Kent (1998) take a visual specification approach to design patterns. The perception is that capturing a design pattern using the current documentation templates, such as a class diagram or a sequence diagram, is not sufficient for abstracting what the pattern is about. This is usually because the pattern’s textual representation and models capture domain-specific information that should be removed from an abstract representation. For instance, in the Observer pattern [Gamma et al. 1995] the sequence diagram model shows two concrete observers for one concrete subject. Similarly, the class diagram for the Abstract Factory pattern shows two concrete factories and two concrete products named ConcreteProductA1 and ConcreteProductA2. These concrete examples hinder the abstract representation of patterns and hence their composition.

Composition should be done at a higher level where details such as the cardinality as well as the concrete names are abstracted and patterns are represented in a way that captures only their essential spirit. Moreover, this abstraction should be supported by graphical notation to enable tool support and facilitate deployment.

Lauder and Kent attribute the reason for these application details that appear in the class diagrams to the absence of a way to express and reason about collections without having to explicitly mention names for the members of collections and their cardinality. As a solution, Lauder and Kent utilize constraint diagrams developed by Kent (1997) together with UML diagrams to represent constrained sets and set members that constitute a notation to describe arbitrary collections.

Using UML and constraint diagram, the UML representation of class is amended with a fourth compartment (other than the traditional UML name, attribute, and operations compartments) to represent abstract instances of that class. The fourth compartment will not carry textual representation of the instances; however, it will carry constraint diagrams that describe instances as sets and relationships between sets using constraint diagram notation. An example is shown in Figure 3-5.

Figure 3-5. A UML class diagram with abstract instances compartment.
In order to purify design patterns from textual details that cause ambiguity, three model representations or layers of abstractions are used to describe a pattern. The three layers are role-model layer, type-model layer, and class-model layer. Role-model is an abstraction of type-model, which in turn is an abstraction of the class-model. The top layer does not use any domain-specific names in describing the pattern; it uses a constraint diagram and extensions to UML models to describe the semantics of the operations rather than their names. It represents the pattern in terms of highly abstract state and highly abstract semantics using a constraint set that captures the essential spirit of the pattern. The type-model is then a refinement of the role-model by adding some specific refinements to the abstract state and semantics such as adding the name of the operations and their concrete syntax. Finally, the class-model is the refinement of the type-model to add application-specific terms such as concrete classes, concrete state names, and concrete method implementations. As an example, the abstract factory is represented in this three-layered abstraction as illustrated in Figure 3.6 [Lauder & Kent 1998].

Figure 3.6. Abstract factory role-model, type-model, and class-model.

This approach is more concerned with visualizing individual patterns than composing patterns. The three-layer abstraction concept is
useful in purifying the pattern from application-specific and domain-specific naming and cardinality, especially when dealing with collections. The composition of design patterns using abstract representation of the pattern could be more useful, since we may abstract concrete names—for example, `ConcreteProductA`. However, the three-layers modeling technique seems visually complex even for simple patterns such as Observer patterns (figures shown in Lauder et al. 1998). Modeling the dynamics of patterns by abstracting the sequence diagram specification and adding the constraint diagram notation could be very complex for most behavior patterns and hence may make the approach difficult to use for pattern composition. Although this work is more concerned with visualizing individual patterns than composing patterns, it can be extended to define composite patterns based on static and dynamic specifications. Applying the approach to composition of patterns is still an immature research area.
Structural Composition of Design Patterns

Structural composition approaches build a design by gluing pattern structures that are modeled as class diagrams. Structural composition focuses more on the actual realization of the design rather than abstraction, using different types of models, such as role models. Behavioral composition techniques, such as roles [Reenskaug 1996, Riehle 1997, Kristensen & Osterbye 1996], leave several choices to the designer with less insight on how to continue to the class design phase. Techniques that consider both structural and behavioral views could be complex and difficult to use. Therefore, the POAD approach advocates a structural composition approach with pattern class diagrams. Constructional design patterns in which a pattern interface can be clearly specified (as discussed in Chapter 4), lend themselves to a structural composition approach.

In the following subsections we discuss several structural composition techniques and contrast these techniques with our proposed POAD methodology. We outline an approach for pattern-oriented design proposed by Ram, Anantha, and Guruprasad in "A Pattern-Oriented Technique for Software Design" (1997). In contrast to our top-down approach, this approach describes a bottom-up process to design software using design patterns. The approach shows how related patterns can be selected; however, it does not clearly show how patterns can be composed. Nevertheless, it gives an example of previous attempts in the literature to develop a systematic process for pattern-oriented software development.

We briefly introduce the concept of software design using design components that can adapt to rapidly changing requirements as proposed by Keller and Schauer in "Design Components: Towards Software Composition at the Design Level" (1998). This work is presented to highlight the importance of the concept of generic design components, where the components are specified in a high level of abstraction. This concept is very much related to the notion of using design patterns as design components. The concept is also different from the concepts of component-based software development, which mainly focus implementation or code-level components as depicted, for example, by component diagrams in UML. Keller and Schauer's work shows how generic design components can evolve to domain-specific design components from which we can obtain code-level components. This approach insures that design quality translates into code quality.

We briefly discuss another approach in the literature on the development of component-based frameworks using patterns. In this work a hierarchical architectural design is developed in which a component can implement many patterns and frameworks, and a pattern or a framework can be implemented across many components. The work takes a structural approach to glue patterns and uses UML interface classes to model interfaces of patterns. UML component diagrams are then used to represent the framework as a composition of physical components that do not have a one-to-one mapping with patterns. We briefly discuss the difference between this work and the POAD approach for pattern-oriented development.

The Catalysis approach created by D'Souza and Wills and discussed in "Catalysis: Component and Framework-based Development" (1998). The approach defines a component-based approach to software development that is heavily based on interfaces at both the design and implementation levels. The Catalysis approach is used to build object- and component-based systems using UML and extensions of UML.

Finally we briefly introduce the newly developed concepts of composition patterns and subject-oriented design [Clarke & Walker 2001; Clarke et al. 2000]. These concepts show an emerging development technology based on the concept of aspect-oriented programming.

A Pattern-Oriented Composition Technique

We start by outlining a structural composition approach that follows traditional OO analysis and design methodologies. Ram, Anantha, and Guruprasad, in "A Pattern-Oriented Technique for Software Design" (1997), describe a bottom-up pattern-oriented approach to design software, Pattern Oriented Technique (POT). They start by analyzing the application in a traditional OO manner. Patterns are then identified and classes are grouped. This approach is not a pattern-oriented development process; it is a traditional OO development process that is augmented with a refactoring technique based on patterns. The steps of the process proposed by POT are summarized as follows:
1. **Identify classes**: The classes are identified based on the requirements specifications using traditional OO techniques.

2. **Identify responsibilities**: Class services are identified to define the behavior of each class. Traditional OO techniques are used to define the service methods in each class.

3. **Identify interacting classes**: Interaction diagrams are developed to define in an abstract sense (i.e., using association or dependency relationships) the interacting classes in what is called class interaction diagrams. Following traditional OO techniques and using UML, collaboration diagrams or sequence diagrams can be used to identify interacting objects that will reflect on relating the classes of these objects in the class diagram using association or dependency relationships.

4. **Identify class groups**: Ram and his colleagues define a class group as a collection of classes such that each class in the group interacts with at least one other class in the group. The number of classes in the group determines the size of the group, which is at least one. These groups can be overlapping; that is, a class may belong to more than one group; hence, a large number of groups is possible.

5. **Identify class group interactions**: The interactions between classes in each class group are identified in terms of service functions. This step would use sequence diagrams in UML to identify the interactions between classes in each class group.

6. **Specify class group interactions at an abstract level**: Interactions between classes in each group are specified in terms of objects, lists of objects, or commands sent from one object to another. For example, an object of a class in a group sends a command to another object of another class in the group, or an object maintains and iterates on a list of objects of another type in the same group. This step is specified to make the group interactions closer to the abstract description of known patterns.

7. **Identify design patterns**: In this step, the intents of design patterns are compared with class group interactions in each group to identify relevant patterns. It is not clear how this comparison should take place, since the intents are usually specified in a textual form, while class group interactions are specified in terms of interactions between objects, as mentioned above. In fact, due to the potentially large number of class groups, this step is very tedious and time consuming. The analysis phase of the POAD process, described in Chapter 8, addresses this problem of selecting the set of related patterns based on the structural and behavioral UML analysis diagrams.

8. **Obtain rough designs**: The design patterns identified in the previous step form a set of rough designs for the class groups. Class diagram refinements are introduced based on instantiations of these patterns.

9. **Evaluate design tradeoffs**: Quantitative measures based on four tradeoffs are derived to assess the tradeoffs of using patterns. These are briefly summarized as follows:

   - **Coding efforts** is a measure of the amount of coding needed for implementing the pattern.
   - **Static adaptability** is a measure of the adaptability of the pattern to a particular context during implementation.
   - **Dynamic adaptability** is a measure of adapting the pattern behavior at run time.
   - **Performance** is a measure of the runtime performance of the pattern in terms of the response time needed to deliver the services expected.

10. **Develop a detailed design**: The patterns are instantiated and glued together by defining the relationships between classes in different patterns. This step in fact is not trivial and is not clearly specified.

The main drawback of this process is that patterns are used late in the process after application classes have been defined. In POT, patterns are used as a refactoring technique. In Chapter 7 we describe a systematic development process for selecting and using patterns in a top-down fashion.

---

**Software Composition at the Design Level Using Design Components**

Component-based software development has become the focus of many recent design and implementation methodologies. Keller and Schauer address the problem of software composition at the design level using design components [Keller & Schauer 1998, Schauer & Keller 1998]. Their project is called Software desirable Properties into the design of Object-Oriented Large-scale software systems.
(SPOOL). The SPOOL project was organized by CSER (Consortium for Software Engineering Research), which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada). The project focuses on the problem of adapting software systems in large telecommunication companies due to rapidly changing requirements. The approach is similar to the POAD approach in creating a design based on well-defined and proven design patterns packaged into tangible, customizable, and composable design components.

Keller defines the evolution of a pattern in a design composition environment along four dimensions:

- Concreteness, which defines how the design component evolves from abstraction to concreteness.
- Specificity, which defines how the design component evolves from being generic to domain-specific.
- Scope, which defines how the component evolves from being a basic component to a more composite form where its internals become of relevance.
- Revision, which defines the versions of the component from its basic form to its current form.

Keller also identifies the limitations of UML modeling of a design component. Components diagrams in UML are used to define code-level components. Patterns in UML are defined as dotted ellipses (collaborations). Keller suggests an extension to UML package diagrams to model patterns and their instantiations.

The structural composition approach proposed by Keller does not define a design process to compose patterns together and does not define a notion for pattern interfaces. Keller uses a package diagram to represent a design pattern as a component. We define visual models that are more expressive because they are based on interfaces. Package representation has several syntax and semantic limitations, as discussed in Chapter 5. The approach in Keller's work (1998), though concerned with a problem similar to ours, focuses on what needs to be placed on a pattern as a design component and little about the procedure of composing components.

Another major concern about this approach is that it integrates two patterns to form a part of the design, and then it integrates another pattern to the design, and so on. This approach loses some benefits of a design component. First, the abstract view of the design is lost; there is no visual view to reflect the high-level structure of the system in terms of patterns. Second, the approach does not use interfaces, which is expected by the user of a component to glue pattern structures together. The pattern visualization tool supporting Keller's approach does not support distinct pattern views that are traceable to class diagrams. Instead, the tool supports covering a class diagram with boundaries on pattern participants. Such technique, also used in Odenthal and Quibeldey-Cirkel's "Using Patterns for Design and Documentation" (1997), is in our view cumbersome and does not support high-level view of the design.

Component-Based Frameworks Using Patterns

The development of application frameworks is now an important topic in modern design and implementation techniques. Larsen (1999) takes a structural approach to glue patterns by mapping the roles and participants of a pattern into actual model elements that fulfill these roles. Larsen uses UML models to glue together components. A component can implement many patterns and frameworks, and a pattern or a framework can be implemented across many components. In his definitions, a framework contains more than one pattern and has several extension points for different applications. In fact, the pattern and framework metamodel shown by Larsen exhibits the Composite pattern, where the pattern is the base class and the framework is the composite derived class with extensions, which is composed of one or more objects of type pattern.

The focus of Larsen's work is designing and delivering frameworks using the concepts of patterns at the design phase and using components at the delivery and implementation phase. In this case, a framework or a pattern can be implemented by several components.

At the design phase of the framework, Larsen uses UML interface classes to model interfaces for patterns. The solution of a pattern as a class diagram is augmented with additional classes as interfaces. These class diagrams of patterns (with additional interface classes) are used to define the framework in one class diagram model. At the delivery phase of a framework, Larsen uses UML component diagrams to represent the framework as a composition of physical components that are not a one-to-one mapping to patterns.

The POAD approach shares the same concept of defining interfaces for patterns. However, Larsen's approach is different from POAD in several aspects:

- Larsen uses class diagrams to represent the composition of patterns' classes. His example shows a class diagram for each pattern (with their interface classes added) and then a class diagram for the framework. POAD uses hierarchical logical models
that hide details that are not utilized directly at a given design-level abstraction. We use additional models for patterns with no interfaces or internal details, then another layer showing interfaces and their associations, and finally the detailed layer showing details and interfaces. Larsen's approach lacks the first two layers.

- **Larsen uses component diagrams for delivering the framework and not for design.** This definition complies with the UML definition of a physical component. Instead, POAD uses patterns as logical design components (similar to packages) rather than physical deliverable components.

- **Larsen's approach does not support hierarchical views of the design.** In POAD we use a pattern as a black box and then zoom into internal details. Larsen uses class diagrams only.

- **The traceability problem is not solved in Larsen's approach.** Pattern participants are lost at the framework design diagrams. POAD models can trace pattern participants' bottom-up and top-down as discussed later in the book.

### The Catalysis Approach

The design of components interfaces is one of the most important steps in the architectural design process. D'Souza and Wills (1998) define a component-based approach to develop software that is heavily based on interfaces at both the design and implementation level. The approach is called Catalysis, which is used to build object- and component-based systems using UML and some extensions made to UML constructs. At the design phase, D'Souza advocates using frameworks as building components, where a framework is "a pattern of model or code that can be applied to different problems" and OO frameworks are "collaborations with a default, skeletal implementation."

We take a similar approach for designing software as a composition of constructional design patterns; however, we distinguish patterns from frameworks. POAD addresses a more specific problem (composition of patterns) and can be used to build such frameworks. Catalysis addresses other issues in developing software, such as composing physical components, distribution of components, and business driven solutions.

Catalysis uses three modeling concepts from UML in order to define components and component interfaces. These modeling concepts are described briefly as follows:

- **Types.** A type models component interfaces by defining the visible behavior of objects. It uses the class construct in UML to specify the attributes and operations of a type. Catalysis clearly distinguishes a type from a class as modeling concepts, since a type specifies only type abstraction, while a class also specifies implementations. The example of this distinction, given in the constructs of a programming language such as Java, is the distinction between an interface construct of Java that corresponds to the type abstraction defined here and the class in Java that implements an interface and hence must include implementations of operations declared in the interface.

- **Collaborations.** Similar to the UML terminology, collaboration defines how interactions between objects or components (i.e., a group of objects) take place using types and actions. Actions are specific invocations of operations defined in a type (i.e., an interface) of an object or a component invoked by a collaborating object or component. The methodology in D'Souza's work treats collaborations as design units. Each design unit defines a design for certain services specified in the system requirements. Collaborations can be composed to define a more complex service or behavior. The collaboration-modeling concept defines the architecture of the application, where every collaboration defines an element of the architecture. This concept of architecture definition is inconsistent with the UML modeling view of collaborations as dynamic realizations of use cases, whereas an architecture is a static modeling view of the components and their interfaces.

- **Refinement.** The third modeling concept used in Catalysis is defining relationships between elements of the other two modeling concepts, that is, between types, between collaborations, or between a type and a class. This concept is based on a relationship between an abstraction and a realization—for example, a type and a class. This modeling concept is very vague and ambiguous, specifically when it applies to collaborations and types. The clarification comes from the concept of specifying collaborations to types in an abstract sense and then defines refinements of these elements as realizations or more detailed specifications of these abstractions; hence, different levels of abstractions can be defined. The UML packages are used to separate these different levels of abstractions. The objective is to permit the reuse of a given abstraction by multiple realizations. A package can be defined as a group of types, collaborations, actions, and even code components.

The modeling concepts described above show that the Catalysis approach is basically a component-based development approach that does not depend on the concept of design patterns as we use in POAD. The above modeling concepts clearly show the difficulty in developing a design using the Catalysis methodology. There is no clear distinction between the dynamic view and the static view. There is
no clear process for developing the design artifacts. Patterns and frameworks are mentioned as reusable abstractions, but no clear techniques are specified on how to integrate these abstractions in the design process.

Composition Patterns, Subject-Oriented Programming, and Aspect-Oriented Software Development

Clarke and Walker have introduced the concept of composition patterns recently in "Composition Patterns: An Approach to Designing Reusable Aspects" (2001) to support designs developed using the Subject-Oriented Design (SOD) model. The SOD model introduced in Clarke, Harrison, Ossher, and Tarr's "Subject-Oriented Design" (2000) has evolved from subject-oriented programming, in which different subjects may be developed and coded to support separate requirements. These requirements, also called aspects, can be overlapping functional or crosscutting requirements. Aspects, just like objects, can be defined at any stage of the software lifecycle, including requirements specification, design, and implementation.

Generally, crosscutting aspects can be defined from design or architectural constraints, or common properties or behaviors, and features. The SOD model is based on the concepts of Aspect-Oriented Software Development (AOSD) evolving from aspect-oriented programming. AOSD is a new technology developed to support the concept of separation of concerns (SOC) in software development. The technology is used to produce modular crosscutting aspects of a system. Examples of crosscutting aspects include logging or tracing of activities, error recovery, and synchronization of activities. The work in AOSD is mainly driven by the fundamental goal of better SOC. Although crosscutting requirements tend to be a significant focus of this technology, the published work [Erlad et al. 2001] also incorporates other kinds of SOC techniques, including the other well-established approaches for developing modules such as OO design and structured design. Hence, AOSD blends support for many different types of modular designs, including block structure, object structure, inheritance, and crosscutting.

In a workshop entitled Advanced Separation of Concerns (ASoC) held at the 2001 European Conference on Object-Oriented Programming (ECOOP2001), and in similar workshops in other major conferences in this field, the presentations clearly show how the SOC concept is providing an added value in addressing difficult requirements in OO development in general and in the problem of design composition in particular.

Crosscutting requirements such as tracing, error recovery, and synchronization are requirements that have impact on and are scattered across multiple classes in a system. The tracing requirement, for example, which involves tracing the entry and exit of selected class operations during runtime, could impact many operations of many classes in the system. Such requirements pose a challenge for developing reusable and maintainable OO designs. The SOD model addresses these requirements by decomposing the design into different design models called design subjects. Each design subject addresses the design of a crosscutting requirement. Composing these design subjects with the rest of the system, however, can also become a difficult and error-prone task.

Clarke and Walker (2001) introduced the concept of composition patterns as a way to develop reusable design patterns for these crosscutting requirements. The composition design pattern addresses the problem of composing the potentially overlapping design subjects with the rest of the system. The concept is based on UML templates, where a UML class diagram–stereotype parameterized package is used to define a reusable design subject. The parameters of this template-stereotyped package can be a list of class names followed by operations or operation signatures of these classes involved in the design subject.

For example, consider the Trace design subject composition pattern [Clarke & Walker 2001]. The Trace pattern is specified as a template package with two parameters, the TracedClass parameter and the _tracedOp() parameter. This template package (or composition pattern) is instantiated by binding it to a design subject called Application package containing Classi with the operation fun() to be traced (see Figure 3-7). This binding produces a design subject calledTracedApplication showing a composition between the Trace design subject and the Application design subject containing the operation fun() to be traced (see Figure 3-8).

![Figure 3-7. The Trace pattern template package and its binding to an Application package][1]

[1]: Clarke & Walker 2001
Figure 3-8. The output of binding the Trace composition pattern with the Application package [Clarke & Walker 2001].

Figure 3-7 shows the Trace design subject template package containing class Trace and class TracedClass, and shows the binding of this
package with the Application design subject package. This binding implies that Classi and its operation fun() in Application will replace the TracedClass and tracedOp(..) specified in the template parameters. This can be expanded to other classes or functions in Application by adding a list of statements to the bind note inside the square brackets in Figure 3-7.

In the TracedApplication design subject in Figure 3-8 the TracedClass and tracedOp(..) were replaced by Classi and the method fun() respectively based on the bind statement shown in Figure 3-7. The method Application_fun() in Classi of the TracedApplication design subject contains in fact the original body of method fun() as specified in the Application design subject, whereas the body of method fun() in the TracedApplication design subject would also contain a call to the traceEntry(String), followed by a call to Application_fun(), and then followed by a call to traceExit(String) method. This behavior is specified in the sequence diagram in Figure 3-8. The sequence diagram shows an interaction model of the behavior of method fun() in the TracedApplication design subject. The parameter of methods traceEntry() and traceExit() is a string containing the name of the entered method. This sequence diagram is the way to specify crosscutting behavior in SOD. It identifies the methods specified in different design subjects that correspond and should be merged. In this simple example, the operation Application_fun() of Classi in the TracedApplication design subject should merge with operation fun() of Classi in the Application design subject. This merge implies that each reference to Application_fun() will be replaced by a reference to the method fun() of Classi in the Application design subject.

The trace example described above can be considered as instantiating a Trace design pattern on an application design. The work of Clarke and Walker in (2001) shows examples of other design patterns, such as the Synchronization pattern in which several functions of a class accessing a shared object need to be synchronized. The Synchronization pattern template package is applied to a design example of a library management application. The Observer pattern template package is also applied to the same example.

The approach described above, as far as pattern-oriented design is concerned, is based on applying design pattern templates to existing designs. Hence, it differs from the POAD methodology, which advocates the notion of developing application designs from scratch using patterns as building blocks.

The use of template packages in this approach is in fact very convenient in terms of specifying the changes to be done when applying a design pattern modeled as a design subject to an application design subject where the pattern can be used. In this case it produces the details of a composition or a single instantiation of the design pattern in the evolving design. This approach in fact can be used in the design refinement phase of the POAD process where domain-specific Detailed Pattern-Level diagrams are developed.
Further Reading

In this chapter we discussed pattern composition approaches that we believe could belong to the behavioral composition or the structural composition categories. We selected some sample techniques, and presented a surface discussion about the concepts and models used by each technique.

Dong, Alencar, and Cowan of the University of Waterloo, Ontario, Canada, developed an approach for behavioral analysis to pattern-based composition \cite{Dong et al. 2001}. They rely upon Prolog and model-checking techniques to analyze the composition and integration of design components. They discuss how to represent, instantiate, and integrate design components and how to find design composition errors. The approach is based on a structural and behavioral specification of the design components based on UML class diagrams and collaborations diagrams, respectively. The class diagrams are used to represent structural evolution of design components that are transformed into Prolog representation. Thus, the design components and their compositions are represented as a logic model. The behavioral specifications are transformed into process algebra representations in which case the behavioral properties of these components are represented as logic formulas. A model checker is then used to verify the properties of the design—that is, to verify that the logic formulas hold in the design model. This approach can be useful when the transformation activity can be automated, which is possible when formal descriptions of patterns are available.

Alencar and his group led a thorough research on the formal specification of pattern behavior, discussed in their latest work \cite{Dong et al. 2001}. Formal specification of pattern behavior as well as pattern relationships are not considered composition mechanisms; however, they can provide assistance in analyzing a pattern composition approach.

A study by Motoshi Saeki of the Tokyo Institute of Technology, Japan, provides a technique to define the behavior of design patterns based on the formal description technique LOTOS (Language of Temporal Ordering Specification). The LOTOS simulation facility is used to check the behavioral consistency of combined patterns. The technique again is dependent on transforming patterns behavioral specifications into LOTOS, which is also related to formal models of patterns \cite{Saeki 2000}.

The above studies clearly show the importance of reasoning about the behavior of individual patterns and the behavior of pattern compositions using a formal technique. Although such techniques are useful in analyzing pattern participants and their interactions and to possibly detect inconsistencies, they do not provide a practical easy-to-use approach to compose patterns that can be used by an OO software designer.

The OOram approach is illustrated in the book by Reenskaug \cite{Reenskaug 1996} with examples from real-world applications. The role models are incorporated in the collaboration diagrams of UML and in the metamodel as classifierRole \cite{UML 2002}. For updates on the integration of role modeling approach into UML refer to Reenskaug's homepage, http://heim.ifi.uio.no/~trygver/.

The notions of role and architectural fragment and their corresponding role and architecture language construct in LayOM that allow for reusable first-class specification of architectural fragments is discussed in Bosch's "Specifying Frameworks and Design Patterns as Architecture Fragments" (1998c). The LayOM language by itself is discussed in Bosch's "Design Patterns as Language Constructs" (1998b). The application of the architecture fragment technique and superimposition to the domain of measurement systems in production lines is discussed in Bosch's "An Object-Oriented Framework for Measurement Systems" (1998a). The superimposition mechanism is discussed in Bosch's "Superimposition: A Component Adaptation Technique" (1999).

For more discussion on modeling patterns using role diagrams and examples from the GoF patterns, refer to Riehle's "Describing and Composing Patterns Using Role Diagrams" (1996). Composite design patterns, their relationships to design frameworks, and details about the process of driving them from concrete applications are discussed in Riehle and Züllighoven's "Understanding and Using Patterns in Software Development" (1996) and Riehle's "Composite Design Patterns" (1997).

In summary, Reenskaug (1996) recognizes roles as analysis mechanisms helpful in developing application designs. Riehle (1997) recognizes roles as composition mechanisms to derive composite design patterns. Kristensen and Osterbye (1996) use roles as first-class conceptual modeling elements from which objects can be derived and therefore classes can be deduced. In their roots, most of the behavior composition approaches stem from the basic concepts of a role, role models, and role diagrams.

Helm and colleagues (1990) describe an approach that uses contracts to compose objects based on their behavior. Contracts provide a formal semantics to describe the mutual obligations between classes of objects. Design by Contract was introduced by Meyer (1992) as an approach to construct software applications. In this approach, the designer focuses on describing the responsibilities of each design
class by formally defining the conditions that should be met prior to using any function or service provided by that class. In addition, a class will provide a formal description of the output of each function or service and how the service changes the application state.

A contract is used to link the function caller (or client) with the function implementation (the contractor). A common way to define a software contract is through the use of preconditions, postconditions, and invariants. Boolean assertions can then be made to determine if a precondition is met before invoking a service, a postcondition is satisfied after the service is invoked (and hence the service is successful), or a class invariant has not changed during the operation. In this case, a contract between the requester and the provider will define the mutual obligations. A client can use a particular service only if the preconditions for the service are met and the class invariants are respected. The provider promises to provide the service and the work defined in the postconditions; in addition, the class invariants will be respected.

Contracts are useful means to formally define the relationship between classes, since they capture all the possible interactions between instances of those classes as well as the runtime conditions required to perform an interaction. They can also be used as means to validate the runtime interactions between objects and discover any violations (interactions not defined in the contract, or pre-condition and post-condition violations).

Jezequel and colleagues (2000) describe how to use contracts to specify the set of design patterns defined in the GoF book. They use the notion of contracts to explicitly describe the intention behind a design pattern in terms of formally capturing the participating objects, their collaborations, and the distribution of responsibilities. They use the Eiffel language to specify a contract for each of the GoF patterns. Although this work mainly focuses on specifying a contract for each pattern that formally defines the responsibilities between pattern participants, the ideas are extensible and useful in pattern composition. For instance, if each pattern is captured by a contract between its participants, can we combine two or more contracts to define a composition of two patterns? Is there a way to present a contract that defines the composition? The answers to these questions can possibly be used to define a behavioral composition technique to compose design patterns. Whereas gluing objects (classes) together using patterns has received thorough investigation in the last decade, gluing patterns together using contracts is still a research topic that has not been exhaustively studied.

In this chapter we also discussed pattern composition approaches that we believe could belong to the structural composition category. We selected some sample techniques and presented a surface discussion about the concepts and models used by each technique.

For more information about the Catalysis approach, refer to D’Souza and Wills's Objects, Components, and Frameworks With UML: The Catalysis Approach (1998), which contains a complete guide to the process and the detailed models and steps. For aspect-oriented design and programming, the latest works by Clarke and colleagues (2000; 2001) provide pointers to its techniques and tools.

The two approaches that we consider closely relevant to POAD are the work by Keller and Schauer, "Design Components: Towards Software Composition at the Design Level" (1998), and by Larsen in "Designing Component-Based Frameworks Using Patterns in the UML" (1999). Keller introduces the concept of software design using design components, which are quite different from the well-known concepts of implementation and physical components. The patterns used in POAD are in fact design components, as discussed in Chapter 4. Larsen introduces the concept of interfaces for design patterns and using those interfaces to develop interfaces for a design framework. The concept of interfaces for design patterns is an important technique that POAD uses to glue design patterns together.

Xavier Castellani and Stephan Y. Liao (1998) propose an application development process that focuses on the reuse of OO application design. Their work presents a process that allows the system designer to create generic applications and reuse them in other application designs in the same problem domain. The approach starts with an existing application, then abstracts design macrocomponents through the abstraction of application-specific design components. The authors use a general definition of macrocomponents (frameworks or patterns) that allows any group of related classes to be considered a pattern.
Part II: Technological Aspects of POAD

In Part II we discuss the technological aspects of POAD. Chapter 4 discusses the role of design patterns as building blocks of software design. We discuss which design patterns can be used with POAD. Chapter 5 introduces the design models that we use to compose design patterns. It also shows how the Unified Modeling Language (UML) syntactically supports these models. In Chapter 6 we discuss the UML support for design patterns. We compare different UML approaches to model design patterns and their composition.
Chapter 4. Constructional Design Patterns as Components

Which Patterns to Use in POAD

Software Components

Specifying a Pattern as a Component

Component Interfaces

Interface Properties

Pattern Interfaces

Summary
Which Patterns to Use in POAD

Design patterns are intended to preserve reusable, good-quality design practices in developing software applications. For this purpose, software designers and developers are motivated to catalog and discover new patterns in various application domains. This has resulted in the growth of a large collection of design patterns in multiple fields. Some of these collections are available in the literature as documents, such as the proceedings of the PLoP and EuroPLoP conferences, as well as the PloPD book series. Others are available in pattern databases in electronic format, such as databases for general-purpose patterns (e.g., the GoF and POSA patterns) or application-specific patterns (e.g., patterns in the telecommunications field).

The collection of widely available literature on patterns encourages application designers to look for techniques to apply and use patterns in software design. While pattern collections are getting larger in size, number, and diversity, classification and categorization efforts are still lagging the huge progress in pattern documentation. There is a large variety of pattern types. Some patterns are usable at the analysis stage, while others are usable at the design or even the development stages. Some patterns provide OO design solutions and constructs in the form of object models, while others are essay-like patterns providing prescriptions, recommendations, or "do" and "do-not" lists of design decisions.

Software designers are motivated to look for techniques to reuse these proven solutions. From an economical perspective, the effort in documenting and mining for design patterns is huge. This effort is paid off when patterns are reused successfully in building new applications. When considering a systematic development approach to reuse patterns in software development, the designer has to consider what type of patterns to use. Usually, some pattern types will be mostly usable with a particular development process, while other types may not. For example, the designer will not use GUI design patterns in developing a telecommunications middleware layer, simply because the patterns are from different domains and their application is not possible.

We are experiencing a growth in the volume of pattern discovery and documentation, and at the same time we are short of classification, indexing, and cataloging approaches and databases. The user of a particular pattern-reuse process has to select the patterns to deploy in his application design. In turn, the provider of a particular pattern-oriented software development process has to narrow down the scope of the patterns to be used with his proposed technique or methodology.

POAD provides a structural approach to use design patterns as building blocks in designing applications. In this context, we have to define which patterns to use in this development process. Are they limited to a specific type? Are there any domain limitations on the applications to develop using these patterns?

For the POAD methodology, we use a category of patterns that we call constructional design patterns. The name is given only to distinguish a set or category of patterns that we use in the POAD methodology. The term constructional is selected because these patterns are used as design components in constructing the application designs; they are the core building block from which the application design is built.

To understand what the word constructional means in the POAD context, we first recall some basic principles of designing software applications. Three decades ago, Parnas used the term information hiding as a core principle underlying software development [Parnas, 1972]. Later, most software development processes used the same principle to build software using units that encapsulate and hide information particular to that unit. Those units usually provide means to access the internals of the unit through what is commonly named interfaces. For example, in OO analysis and design, a class hides design decisions and data relevant to that particular class and offers interfaces (methods) to access the class internal state. Similarly, in the POAD methodology we think of a constructional design pattern as a unit of building software design that encapsulates information; it encapsulates a solution to a frequently recurring design problem, it hides lower level design decisions, and it offers interfaces to other design artifacts. In this sense, a constructional design pattern becomes a design component with interfaces. Specifying a pattern as a design component leverages the interest in a pattern to a higher design level that hides later design details yet preserves consistency with lower levels. The terms design component and constructional design patterns are more precisely defined in this chapter.

To summarize the role constructional design patterns play in the POAD methodology, we recall the basic principles of structured analysis and design (SA/SD) and OO analysis and design (OOAD) methodologies. SA/SD uses function modules (procedures) as application building blocks; modules are composed to provide an application design. OOAD offers classes as building blocks; classes are connected (using some class–class relationships) in a class diagram to provide a design of the application. In POAD, we use constructional design patterns as building blocks; patterns are linked, connected, and refined to develop application designs.
The constructional design patterns used in POAD are OO design patterns that abstract a structure of collaborating classes. To this extent, the application design is built by gluing together these construction fragments and defining dependencies and collaboration between participating patterns, as discussed in Part III. Usually, a design pattern is documented for white box reuse. This means that when a pattern is reused in the design of a new application, it will be instantiated; generic names will be substituted with domain-specific names for the classes and methods inside a pattern. In POAD we define interfaces for constructional design patterns such that they can be manipulated as black boxes at a high design level (when we are not interested in how the pattern solves a particular problem, but we are interested in knowing that it solves the problem at hand), and we defer the instantiation details to a lower level.

Whereas the problem of using design patterns as components in building application designs is a hard problem, the problem of using specific application frameworks is even harder. In the former, general-purpose patterns are used as building blocks, and at a later design phase their application-specific details are added. Each of these building blocks has few classes. Specific OO frameworks consist of larger numbers of classes. It is a hard problem (that should be dealt with separately) to define interfaces for the large population of classes in a framework. It is also difficult to find means to combine frameworks that are designed from a large number of classes [Fayad & Schmidt 1999].

In the following sections we discuss some concepts related to using patterns as building blocks of application designs. We first discuss the concept of component-based development and how ideas from that discipline are related to or differ from the POAD pattern composition approach. We identify the category of patterns that we use in POAD. We define the terms constructional design patterns, design components, and pattern interfaces. Those definitions will be used throughout the book.
Software Components

Component-based development is a widely growing engineering discipline that refers to the lifecycle of developing software components and developing applications using software components. Component-based software engineering is a special case of software engineering for and with software components [Mili et al. 2001]. It is a software engineering discipline that advocates the development of software systems as the integration and assembly of software components. With the growing literature on component-based software engineering (CBSE) [Szyperski 1998; CBSE99; D’Souza 1998], the term software component has become overloaded with multiple definitions. In software engineering the various development phases and richness of information of a software unit have caused the emergence of multiple definitions of a component with little consensus on what it practically means. Originally, computer scientists and engineers referred to any building block of software, such as specification, code, or design, as a software asset. Recently, many researchers and practitioners adopted the term software component; each has a different interpretation of the word. The multitude of definitions originates from the multiple dimensions along which we characterize component features. These dimensions include, but are not limited to, distribution, packaging, execution, composition, deployment, encapsulation, and client/provider relationships.

Ideal software components, as envisioned by researchers and practitioners, are self-contained, fairly independent concrete realization, require little or no customization and provide well-defined services for the application in which they are integrated [Cox 1990]. Moreover, components could be built inhouse or acquired commercially. Commercially acquired components are often called commercial-off-the-shelf (COTS).

In CBSE we may make the assumption that components are binary black box units of software. Executables or libraries (static or dynamic) fall under this category. Alternatively, we may assume that components are deliverable pieces of code developed by one party and used by another party. Usually, the component user will not change the source code, not only because it is difficult to change code that is not developed by your own development team, but also due to contractual agreement with the component supplier (vendor).

In POAD we consider a different type of component that we call design components, which are software design fragments rather than pieces of compiled binaries or pieces of code. In the following sections we define the design components that we use throughout the book. We also define the type of patterns that we use in the POAD methodology.

A Design Component

A design component in the POAD methodology has the following characteristics:

- A design component is a software design fragment.
- A design component is represented using a design notation and delivered as a design model. For instance a design component in OOAD can be represented using a class model of one or more classes represented in UML notation. In SA/SD a design component can be represented using a dataflow model of one or more functions (procedures) represented in dataflow notation.
- A design component is deployable at the design time, such as when developing the class diagram models of an OO application.
- A design component is a white box component. A design component encapsulates a design advice and its solution model that could be deployed in designing the application. During the application of that design fragment, the designer and the developer need to have access to the details of the design for instantiation and implementation purposes.
- A design component has well-defined interfaces. Interfaces are used to glue and integrate design components together and with other design artifacts.

It is clear from these characteristics that a design component is not a black box unit of software as mostly referenced in the CBSE
community. This is because during the design of software applications, the designer works with design fragments, the internals of which he should have access to, whereas in CBSE the application integrator is mostly concerned with gluing executable pieces together. In POAD the output of applying the methodology is an OO design, not an executable application; the same is the case for OOAD methodologies. However, in CBSD the output of integrating components is an executable application.

Design components are used as units for design composition by instantiation, specialization, and assembly. In addition to the above properties, a good design component has some important characteristics that allow it to be used in design composition and integration. A design component is

- **Composable.** A design component has interfaces by which it is glued together with other design components. A design component has internals and externals. Internals are defined in terms of the internal structure and behavior models. Externals are defined in terms of interface models, that is, models for composition with other components.

- **Customizable.** A design component can be customized to allow selection between tradeoffs at lower design levels.

- **Persistent.** The internals of a design component are preserved after instantiation and are traceable throughout various design levels.

### Constructional Design Patterns

A common definition of a design pattern is a design solution to a frequently recurring design problem in a particular context. This is a very general definition that could encompass a lot of pattern categories. In the POAD methodology, we use a specific type of pattern that we call a constructional design pattern. A constructional design pattern is a design pattern with additional constraints that allow for composition and integration. A constructional design pattern provides a design solution that is modeled as an abstraction of a common design structure; the design structure is represented using a UML class diagram. Therefore, a constructional design pattern

- Is an OO design pattern,
- Has interfaces for composition and integration, and
- Its solution has a class model; that is, it provides a solution as a structure of collaborating classes.

Constructional design patterns are design components that can be glued together at high design level. This composition defines the application's overall solution structure. The name constructional design patterns is given because these patterns are used in constructing the structure of the application design using their class models. The internal details of the constructional design pattern are hidden at high design levels (pattern views) and are traceable to lower design levels (class views).

As an example of a constructional design pattern, consider the Composite pattern \[\text{Gamma et al. 1995}\]. The Composite pattern has a class diagram as its solution structure, which contains **component**, **leaf**, and **composite** classes. The **component** class is considered the interface of this pattern.

As examples of patterns that are not constructional design patterns, consider the Pipes and Filters or the Layers patterns \[\text{Buschmann et al. 1996}\]. These patterns do not provide a solution as a class model. The solution is an architectural idea of how to organize the application. However, we cannot generalize and claim that architectural patterns are not constructional, because a Blackboard pattern \[\text{Buschmann et al. 1996}\] is constructional: It has solution structure as classes, and we can define its interfaces.

Constructional design patterns should not be confused with structural patterns defined in the GoF book \[\text{Gamma et al. 1995}\]. A structural pattern is a composition of classes or objects. Structural class patterns use inheritances to compose classes, while structural object patterns describe ways to assemble objects. Constructional design patterns have no restrictions regarding the way to construct the pattern. However, the presence of a class model and the possession of the pattern interfaces are essential in any constructional design pattern.

To qualify a software artifact as a component, it must possess interfaces. POAD uses constructional design patterns as design components, and hence they should possess interfaces. Pattern interfaces are essential to glue patterns together at the high design level without exposing internal parts of the pattern that are not directly utilized at that level of abstraction.

In the following sections we discuss how to specify patterns as design components and how to specify the necessary pattern interfaces.
that allow for composition and integration.

[Team LiB]
Specifying a Pattern as a Component

A pattern can be described and specified in a variety of forms. A precise specification of a pattern is the basis for pattern selection (matching patterns against a set of application requirements) and is the basis for pattern composition (building applications using patterns). We classify techniques to describe a pattern into three categories: recipe, formal specification, and interface specification. Each is briefly described in the following sections. Since we use patterns as design components in the POAD process, we are mostly concerned with the interface specification.

A Recipe

A recipe is an informal description of the pattern that helps application designers to understand the pattern without overwhelming analytical and theoretical details. Several pattern formats have been used to describe and document patterns; the GoF [Gamma et al. 1995], POSA [Buschmann et al. 1996], and Alexandrian [Alexander et al. 1977] templates are examples of pattern formats that can be considered a recipe (prescription/recommendations) for applying a pattern. Essential elements of a pattern recipe include

- The context in which the problem is often incurred.
- The problem solved by the design pattern.
- Forces influencing the selection of the pattern solution.
- The solution to be used for the problem at hand.
- The consequences of applying a pattern.

The recipe description of a pattern is the most convenient representation that helps pattern users understand the pattern. When it comes to composition with other patterns, the recipe is not sufficient to guide the integration process.

Formal Specification

Practitioners usually find the process of formally specifying a software component as a difficult task with little short-term return on investment. On the other hand, researchers are often tempted to validate a software component specification using mathematical and logical techniques. There are few research initiatives that are concerned with formally specifying design patterns [e.g., Eden et al. 1996; Alencar et al. 1995]. This effort is motivated by the need to provide a more precise mechanism for describing pattern and for improving pattern understandability. Specifying patterns in a well-founded formal method helps designers to reason about several solution issues and hence improves the understandability of the pattern and guides the process of improvement. However, many experienced systems designers confront the idea of using formalism in patterns because patterns encapsulate mental reasoning decisions beside their technical solutions that are difficult to capture using formal techniques.

Interface Specification

We are particularly interested in another type of specification for patterns, which we call interface specification. In POAD constructional design patterns are used as design components; therefore, we are interested in pattern interfaces for the purpose of composition with other patterns and with other design artifacts. Specifying interfaces for a pattern is an essential step toward its visual specification. By
visual specification we mean a consistent visual representation of the pattern that highlights its interfaces to other design artifacts. A constructional design pattern has a predefined structure of collaborating classes. At high-level design, it is essential to be able to visualize application designs as compositions of constructional design patterns. At that level, the application designer is not concerned with how individual patterns perform their required task, but he glues together these components that interact to achieve the overall application functionality. In order to apply this glue, the designer should understand which parts of the pattern are crucial for integration—that is, its interfaces.

In the following sections we briefly discuss current approaches to define interfaces for software components. Then, we define some of the attributes that characterize a component interface. We define pattern interfaces and describe their properties.
Component Interfaces

Before we select the type of interfaces that are suitable for constructional design patterns, we first draw from previous experiences in defining interfaces for software components. To compose components together, we need to glue them in an architectural context. To be able to glue components, we need to define their interfaces as viewed by the rest of the components in the system. Component interfaces heavily depend on the nature of the component; a component could be a code fragment, an architectural abstraction, or a design unit. A code component has interfaces that are mostly dependent on the programming language used in the component implementation. Architectural component interfaces are more abstract and are usually represented in a formal or specification language often called an Architecture Description Language (ADL). In the following section we summarize some techniques used to define component interfaces, and then we illustrate how these techniques impact the selection of pattern interfaces.

Module Interconnection Languages

Module Interconnection Languages (MILs) are used to describe large program structures independent of the programming language used to implement that structure. Several MIL languages have been developed and used by researchers and practitioners [for example, Perry 1987; Reiss 1990; Purtilo 1994]. The objective of these languages is to increase the vocabulary of connection and increase the semantic content and checkability of module interfaces. For instance, Lam and Shankar, in “A Theory of Interfaces and Modules—Composition Theorem” (1994), distinguish important aspects of module interfaces, such as differences between interfaces that are service providers (servers) and interfaces that are users (clients). Diaz and Neighbors, in “Module Interconnection Languages” (1986), provide a survey of these languages.

Many of these languages define program structure by defining bindings that connect various program constructs, such as data structures and functions. MILs are not usually concerned with what the system does or how the individual modules implement their functions (detailed design information). Instead, they are used to define relationships between program fragments (modules). Such languages, with emphasis on programming aspects, have several limitations when used to describe a system architecture—that is, the structure of the system as a composition of components and connectors [Allen & Garlan 1997]. Though superseded by recent advances in architecture description languages, MIL languages are useful for two reasons:

- Architectural components will eventually be mapped into one or more pragmatic components, and hence architectural interfaces will be translated to programmatic ones at a later development phase for which MILs are suitable.
- For component-based software development using COTS components, the COTS component interfaces will usually be expressed as programmatic interfaces, and hence MILs will be more applicable than high-level abstract languages.

Interface Definition Language

The Interface Definition Language (IDL) is part of the Common Object Request Broker Architecture (CORBA) specification. It is a specification language that is standardized by ISO as ISO/IEC 14750. It defines the capabilities of a distributed service along with a common set of data types for interacting with these services. IDL has some properties that are key principles in making it an interface language. These properties include its programming language-independence, hardware-independence, and the ability to specify distributed services and complex data types.

IDL allows component interfaces to be defined independently of their implementation. After an interface is defined in IDL, the interface definition is used as an input to an IDL compiler to produce an output that is programming language-dependent. The output can be
further linked with the component implementation. This level of indirection provides a distinction between abstract interface definition and interface definition that is programming language-specific.

A client component (or application) may inquire about another component's interfaces at runtime. The client component is provided the facility to construct a request dynamically depending on the interface requirements of the server component. To accommodate such component-to-component services, Dynamic Invocation Interface (DII) is defined within the CORBA environment. The DII is supported by an interface repository, which is defined as part of CORBA. By accessing information in the interface repository, a client is able to retrieve all of the information necessary about a component interface to construct and issue a request at runtime.

Interfaces for Object-Oriented Components

The UML adopts an OO definition of component interface as "a collection of operations that are used to specify the service of a class or a component" [Booch et al. 1999]. Interface operations are further traceable to the implementation parts of the component; these implementation parts are programmatically methods of specific classes.

Contracts [Helm et al. 1990] are used to add another dimension to OO interface—that is, sequence of interactions. Contracts define the specification of interfaces between OO classes and explicitly specify interactions among groups of objects. A contract specifies participants in the contract, the functionality provided in the contract, and invariant conditions. The SELECT approach [Allen & Garlan 1998] uses the idea of contracts to define interfaces.

In OO designs, interfaces can be model elements by themselves. For example, an interface can be a collection of operations that are implemented as class methods. However, interfaces can also be specifications, such as contracts that are implemented by several model elements: classes, associations, interactions between class instances, and so on.

Application/Platform Interfaces

From a component-based engineering perspective, we identify two different classes of component interfaces: application interfaces and platform interfaces [Yacoub et al. 1999].

Application interfaces define the import and export relationship with other components (or the middleware) with which the component interacts. A set of exported interfaces represents the functionalities that the component provides. A set of imported interfaces represents the functionalities that the component requires from other external components. We term these interfaces horizontal channels because they specify the interaction with other peer components irrespective of the platform on which they run. The horizontal channel allows us to identify the structure of messages sent and received from other components, timing issues as related to requests coming in and out of the component, incompatibilities in data format and types, and incompatibilities in the message protocols.

Processor, memory, communication equipment, and probably other hardware support a component execution. The interaction of a component with these elements constitutes the platform interfaces. This type of interaction is as important as interaction with other software components. It determines the portability of the component. The platform also includes the operating system and its supported services. This layered approach helps the designer in specifying and designing components that are independent of programming languages and operating systems. This interface layer is also called vertical channel because it shows vertical interaction with lower layers of hardware, not with other peer components. This type of interface is essential for special applications (embedded systems, for example) in which 20 to 30 percent of safety-related errors are related to these interfaces [Heimdahl et al. 1998]. Platform interface examples include operating system, hardware platform, communication channels (and protocol stacks), and interpreters and compilers (if required to interpret/compile the component code for a specific platform).
Interface Properties

We consider some general properties when defining interfaces for constructional design patterns. This collection is derived from an analysis of various techniques used to describe component interfaces in general and OO interfaces in particular. These properties help us understand the kind of interfaces we are looking for in constructional design patterns. The set of properties include the type, role, nature, dynamics, descriptions, and multiplicity of interfaces.

Type

Interfaces can be functional or referential. Referential interfaces to a particular component (the provider or server) mean that a client (or requester) component has reference to the provider component, but we do not know at this design stage any details about the usage relationship. Referential interfaces are useful for building the design structural views, since we are able to determine which component uses the other components. Functional interfaces define one or more particular services that a component requires or provides. Functional interfaces are useful in building the behavioral views of the system, since we know at this design stage what services one component needs from another.

Functional interfaces specify the functionalities (services) provided by the design component and the functionalities required from other design artifacts. This can generally be classified as

1. Services. These are the operations provided by the design component that can be invoked by other design artifacts requesting a particular service.

2. Events. These are actions taken by the design component as part of its processing progress; they represent invocations to other operations that belong to other design artifacts.

Referential interfaces represent the knowledge of a design component about the existence of other design artifacts, such as other patterns or classes. In the OO design context, these interfaces could be

1. Class References. A design component interface can be a reference to another class in the design. A typical programmatic reference can be a pointer or a reference in the programming language constructs.

2. Pattern References. A design component interface can be a reference to another design component in the design. In POAD we use design patterns as components, and hence there could be references to other patterns. At the time of writing, there is no programmatic support for pattern references because patterns are not supported as data types in any programming language.

Role

The role classification emphasizes the client/server relationship and hence explicitly distinguishes the interfaces required and provided by a component. We term this classification as role classification because it distinguishes the role that is played by a design component (client/server) with respect to other design artifacts:

A Requester (Client) Role: The design component plays the role of a client. The design component requires services from other components to complete its functionality. Examples of such interfaces include
• References to other classes and design components (this is also a referential interface as classified by the Type property of an interface).

• An operation call in which the design component hosts the calling operation (this is also a functional interface as classified by the Type property of an interface).

A Provider (Server) Role: The design component plays the role of a server. These are the parts of the design component that are referenced by other design artifacts. Examples of such interfaces include

• Classes of the design component.

• Methods of a class that are called by the other design artifacts. (This is also a functional interface as classified by the Type property of an interface.)

Nature

An interface can be abstract (i.e., cannot be instantiated) or concrete (i.e., can be instantiated). In most situations, interfaces are abstract—for example, the interface construct in Java or abstract classes in C++. In other situations, a designer can use a parent class in an inheritance tree as the interface for that tree. In such cases, the interface is concrete; it provides default implementation for the interface operations.

Dynamism

Static interfaces are predefined, known, and published for users at the design time (for example, CORBA IDLs and Java interface classes). Dynamic interfaces are not specified for users at design time; instead, they are interrogated (inquired) by the calling component at runtime (for example, CORBA DMI).

Description

An interface is characterized by the way it is described. An interface can be described using a signature or a behavior. The signature part describes the names and parameters, while the behavioral part specifies how the component reacts to invocations of its interfaces (but not how it implements this reaction) [Bergner et al. 1998].

Multiplicity

A component can have multiple interfaces, all of which are valid interfaces to the same component. According to the context in which the component will be used, the component may reveal one or more of its interfaces. For example, in Java a class can implement one or more interfaces, and according to the application context using that class, one of those interfaces will be used.
Pattern Interfaces

In POAD we utilize constructional design patterns as design components. We use these components to build the logical model views, as discussed in Chapter 5. To qualify a software artifact as a component, we must define its interfaces. Patterns are used here as high-level design components for constructing OO systems. The interfaces for these patterns must conform to constraints imposed by technology dependency (OO paradigm) and the level of abstraction (architectural/high-level design). The definition of interfaces for patterns is gaining the interest of several researchers and practitioners. Larsen (1999) uses stereotyped interface classes to model pattern interfaces at the class diagram level. Larsen defines interfaces for some popular patterns, such as the Party and Account patterns.

We need to define interfaces for constructional design patterns because they will be glued together at a high level of abstraction. Pattern interfaces are important to

- Hide pattern details that are not utilized at high abstraction levels.
- Distinguish parts of the pattern crucial for integration and composition.
- Provide the flexibility to have different implementations of the pattern internals.

Pattern interfaces are not implementations or module interfaces because they are defined and used at the design level. Defining interfaces for a pattern does not mean that the pattern will be implemented as a black box unit of code. Pattern interfaces are intended to facilitate composition of those design artifacts at the design level. At later design and implementation levels, the internal design of the pattern will be integrated and merged with other design artifacts. Therefore, it is not suitable to use MIL or ADL to describe a pattern as a back box implementation unit.

For constructional design patterns, the internal design of those patterns is expressed as OO class diagrams. Therefore, it is reasonable to consider constructional design pattern interfaces as OO interfaces. However, the definition of interfaces in UML is limited to operations. A pattern represents a higher design level construct than classes, and hence its interface can be a participant class of the pattern itself rather than a particular operation.

Pattern interfaces are application interfaces (using the application/platform classification discussed in the previous section) because they are used to glue patterns together. They are not platform interfaces because they are used at an abstraction level that does not mandate a specific platform implementation. Therefore, pattern interfaces are technology dependent (OO) but are platform independent.

Figure 4-1 illustrates a general schema of a constructional design pattern with emphasis on interfaces. We distinguish three main sections: the pattern informal description, interfaces, and internals.

Informal Description. Many designers use patterns as mental exercises during the design phase in which they know a solution and they debate the tradeoffs associated with that solution. Hence they believe that patterns should be preserved as solution prescriptions to design problems. The pattern informal description encompasses the context of pattern usage, intent, problems, forces, and consequences of applying a pattern. This prescription is usually captured by one of the pattern templates (Gof, POSA, or Alexandrian templates).

Interfaces. Pattern interfaces define how the pattern interacts with other design artifacts. We define pattern interfaces as interface classes and interface operations. Interface classes are realized by classes of the pattern's internal class model. Interface operations are realized by the pattern's internal classes that offer methods implementing these interface operations. Interfaces are generally

Required Interfaces. A required interface includes referential interfaces to other design artifacts (references to other classes or patterns) and internal methods in the pattern classes calling other design artifacts.

Provided Interfaces. A provided interface includes the internal elements of the pattern that can
be invoked from other design artifacts. This can be referential, such as classes, or functional, such as methods of a class inside the pattern.

*Internals.* This section represents the internal structure of the design component. In an OO context, the structure is represented in terms of a structure of classes (a class diagram) and collaborating objects (object interaction diagram).

**Figure 4-1. A pattern template with emphasis on interfaces.**

To ease traceability from high-level abstraction in terms of interconnected patterns, to lower level views of collaborating classes, we use *interface classes* and *interface operations* as pattern interfaces. These interfaces are used in the logical model views in Chapter 5, specifically the *pattern-level with interface view.*

We can determine the properties of pattern interfaces using the classification dimensions described in the previous section.

- From a type perspective, interface classes of a constructional design pattern are referential interfaces and are used for composition with other design artifacts. Interface operations are functional interfaces and are used for understanding interactions with other design artifacts.

- Interface classes for constructional design patterns can be abstract or concrete, and interface operations can be virtual or concrete.

- Interface classes (operations) can play the role of client or server (provided or required) interfaces.

- At lower design levels, interface classes are used to establish class association relationships between classes of different patterns, while interface operations help in the construction of object-collaboration diagrams.

- Since constructional patterns are design artifacts, their interfaces are static.

*Table 4-1* summarizes the interface properties for constructional design pattern interfaces.
<table>
<thead>
<tr>
<th>Property</th>
<th>Interface Classes</th>
<th>Interface Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Referential</td>
<td>Functional</td>
</tr>
<tr>
<td>Purpose</td>
<td>Compositional</td>
<td>Collaboration</td>
</tr>
<tr>
<td>Nature</td>
<td>Abstract/Concrete</td>
<td>Virtual/Concrete</td>
</tr>
<tr>
<td>Role</td>
<td>Client/Server</td>
<td>Provided/Required</td>
</tr>
<tr>
<td>Traceable to</td>
<td>Class Association/Dependencies</td>
<td>Object Links</td>
</tr>
<tr>
<td>Dynamics</td>
<td>Static</td>
<td>Static</td>
</tr>
</tbody>
</table>
Summary

We defined specific types of patterns that we use in the POAD methodology. Those patterns are OO design patterns, have interfaces, and are modeled using class diagrams. We refer to those patterns as **constructional design patterns**. We discussed the role that constructional design patterns play as design components. Since interfaces are crucial for software composition tasks, we summarized techniques to define component interfaces from which we derive some properties that we use to classify component interface. We defined constructional design pattern interfaces as interface classes and interface operations, and discussed their properties. The definition of interfaces for constructional design patterns is essential for the integration and composition models and for processes discussed in the following chapters. For another discussion on pattern interfaces, refer to G. Larsen's "Designing Component-Based Frameworks Using Patterns in the UML."
Chapter 5. Visual Design Models

All software development methods should be supported by a set of models. Models capture the outcome of each step in the software development process. Models are useful for documentation; they serve as a communication means between various parties involved in the development lifecycle. A good development process provides support for feeding the output models from one phase into the other. Models produced in one phase should make use of and build upon models produced in earlier phases.

In this chapter we discuss the design models used in the POAD methodology. Those design models support the structural composition of constructional design patterns at various phases of the POAD process, as discussed later in Chapter 7. We do not invent new design model constructs; instead, we utilize existing modeling capabilities as made possible by the UML notation and the UML extension mechanisms. Pattern composition approaches that do not adhere to UML specifications are not likely to survive. UML is the de facto standard for modeling OO applications, including pattern oriented ones.

In the following sections we illustrate three basic models that we use in POAD. How they are used in the development process is discussed in Part III, which describes the POAD process, and Part IV, which describes some examples and case studies. So, while reading through this chapter, you may wonder how these models fit together and how to create them. The answers will follow, but for a quick summary, you can jump to Chapter 7 for a peek.
Pattern Composition Models

Pattern visualization and pattern composition visualization are essential to fully comprehend the role that patterns play in the software development process. Pattern visualization refers to the models that we use to capture the internal design of a pattern. Pattern composition visualization refers to the models that we use to capture the integration and composition of a set of design patterns.

For pattern visualization, UML class diagrams are used to capture the internal structure of a pattern. UML collaboration (interaction) diagrams are used to capture the internal behavior (dynamics) of a pattern. Pattern composition visualization depends on the way patterns are utilized in the software development process. Each pattern composition approach (e.g., see Chapter 3) defines its models for pattern composition; some use UML and others have their proprietary models. Precise visual modeling of pattern composition is required for the development of CASE tools that support the process of designing with patterns. Automated support for design composition models provides the designer with the environment that is required to operate at different levels of abstraction. At a high level of abstraction, patterns are used as building blocks while the internal design of the pattern is revealed at lower abstraction levels.

In POAD we construct visual models that capture the structural composition of patterns. As a way of capturing pattern composition, the models used in POAD provide support for visualization of patterns at various levels of granularity and details. These visual models should contain more coarse-grained models than class diagrams. Class diagrams are suitable for capturing the internal design of a pattern. When it comes to integrating design components (patterns) that are coarser than classes, a set of coarse-grained models is needed. The refinement of these models produces the class diagram of the application.

POAD models make use of the principles introduced in the Hierarchical Object-Oriented Design (HOOD) [Hood 1993; Robinson 1992]. In HOOD objects are not only fine-grained design elements but are used as containers to encapsulate other objects. POAD uses two important concepts from the HOOD design models:

- **Hierarchy.** HOOD provides models that capture hierarchy of objects. Each object may contain other objects in a hierarchical fashion. As a pattern composition mechanism, the POAD process requires models that provide this hierarchical design not in the fashion that patterns contain patterns but instead in the fashion that patterns contain classes that could be hidden at a high design level. POAD models support a coarse-grained view of the design as a composition of constructional design patterns while hiding pattern internals. At later design phases, POAD models provide support to express the internal design of individual components (patterns).

- **Connectivity and Traceability.** HOOD models offer a connectivity mechanism between interface objects and internal objects. Coarse-grained objects will have connections with other objects through a set of interfaces. The interfaces of the coarse-grained object are then traced to its internals. This connectivity mechanism makes the HOOD models usable at different levels of abstractions with appropriate traceability between the models used at various levels. In POAD we need a similar approach. A constructional design pattern is composed of several collaborating classes. From a compositional perspective, the pattern is the container and its constituting classes are the content. The interfaces provided by the pattern are implemented by the internal classes of the pattern. Therefore, POAD models should support traceability between pattern interfaces to pattern internals.

We follow some useful guidelines in defining the POAD models. These guidelines are influenced by the general characteristics that we should seek in a good model as influenced by modeling with UML [UML 2002]. These characteristics include the following:

1. **Model elements are chosen to serve a purpose.** Model elements should be as close as possible to the mental building blocks that we use to solve design problems. Design patterns are mental building blocks that provide solutions to frequent design problems, and hence they qualify as building blocks of a model. POAD models should provide first-class constructs that represent patterns as units for building designs.

2. **Models tend to be hierarchical.** Hierarchy is a technique to manage the system complexity. Most software systems are complex enough that the flat design is not suitable. Hierarchy captures design ideas at various levels of abstraction. POAD models should provide support for hierarchy.

3. **Models should be exchangeable.** Models should integrate with other models in the same field. For instance, UML is the common modeling language in the OO field. POAD models should integrate with and make use of UML modeling constructs.
4. Models serve other models. Design models that we use at one phase of the project should be useful in other development phases. A design model that is developed once and never used again is a weak model. Models should be linked with each other at various development phases and should complement one another.

POAD has three design models for composing patterns. Each model is presented in a design diagram:

1. **Pattern-Level** model. This diagram expresses the design in terms of interfacing patterns and identifies pattern dependencies. This is a high-level view that captures the system design as a composition of design patterns.

2. **Pattern-Level with Interfaces** model. This model is a refinement of the Pattern-Level view using pattern interfaces and explicitly defining relationships between interfaces. At this design level, we are concerned about the relationship between pattern interfaces.

3. **Detailed Pattern-Level** model. At this level, the internals of the pattern and the connectivity between interfaces and internals are revealed.

In the following discussion, each model is described according to its

- **Schematic diagram**: The design artifacts used in the model and how they are interconnected.
- **Relationships**: The semantic relationship between any two artifacts in the model.
- **Design decisions**: The important decisions made by the designer.
- **UML syntax**: How to use UML models to syntactically construct POAD models.
Pattern-Level Model

The Pattern-Level model represents the system (or a subsystem) as a composition of design patterns. This is a high-level structural view of the system. The POAD user at this level is concerned with representing the system as patterns and capturing the relationship between various patterns.

Schematic Diagram

The schematic diagram for the Pattern-Level model represents the patterns that the designer uses and the relationships between them. When the designer selects to use a specific pattern in an application, we say that the designer created an application-specific pattern instance, or pattern instance for short. At this level, the representation of a pattern instance captures two things: the type of the pattern instance and the name of pattern instance.

**Type of Pattern Instance**: Each pattern instance that we use in the design has a type. We use type to refer to the well-known and documented name of the pattern. For example: Observer, Factory, and Strategy [Gamma et al. 1995] are pattern types.

**Name of Pattern Instance**: This is the name of the application-specific instance of a pattern as given by the designer. There are several design situations in which the designer may use two patterns of the same type. For example, the designer may want to use two instances of the Observer pattern in the design of a feedback system. The first instance is used to design the part responsible for observing changes in the plant to be controlled. The second instance is used to monitor the changes in the user input data. In this and similar situations the designer should distinguish which observer is used and in which situation. The instance name is used to distinguish each observer. For example, PlantObserver and UserDataObserver are pattern instances of the type Observer.

A high-level representation of the model that captures these design rules is shown in Figure 5-1 (and later in UML notation). In this diagram, patterns are represented as rectangles labeled with Pattern Instance Name and Type.

**Figure 5-1. A schematic diagram for the Pattern-Level model.**

![Pattern-Level Model Schematic Diagram](image)

Relationships

At this level, dependency is the relationship between patterns. A pattern dependency indicates a semantic relationship between two
patterns: a situation in which a change to the source pattern may require a change to the target pattern or in which a pattern delegates responsibilities to another. A pattern dependency is a uses relationship. Pattern dependencies are further refined at later design phases to become associations between interface classes of two dependent patterns.

**Design Decisions**

The design decisions made at this phase include

1. Selection of design patterns from a pattern repository. Part III discusses some guidelines for the selection criteria and the development process.

2. Defining pattern dependencies. The designer decides how a pattern will use another pattern and the type of the relationship. In most of the case studies that we discuss in Part IV, a uses relationship between patterns is common practice.

**UML Syntax Support**

Schematically, the Pattern-Level view resembles UML package diagrams. Patterns are represented as containers (holders) that encapsulate design solutions (in terms of a class diagram) to some design problems. POAD uses packages to represent constructional design patterns. The name of the package is used to represent the pattern instance name. In order to distinguish packages that represent design patterns, we use UML extension mechanism. Packages that represent patterns are stereotyped using the type of the pattern instance.

Relationships between packages are often represented in UML as access or generalization relationships. In POAD we can use the UML extension mechanism to stereotype a package-to-package relationship as a uses relationship between two pattern instances. Figure 5-2 illustrates the Pattern-Level model using UML syntax.

**Figure 5-2. A Pattern-Level model using UML syntax.**
As an example, consider an application in which the designer has decided to use the Reactor pattern (Schmidt 1995b) and the Composite pattern (Gamma et al. 1995) to design a reactive system that has a set of reactors to some specific input events, and some of those reactors are hierarchical and composite in nature. The designer gives the pattern instance name MyReactors to the Reactor pattern instance and the name CompositeReactors to the Composite pattern instance. Figure 5-3 illustrates the Pattern-Level diagram for this simple example.

Figure 5-3. An example of Pattern-Level diagram in UML.
Pattern-Level with Interfaces Model

The Pattern-Level with Interfaces model takes the design into the next level of details. The purpose of this view is to explore the details of the relationship between the patterns used in the Pattern-Level view. The uses relationship between patterns is further refined. In order to refine this relationship, we need to reveal the interfaces offered by each pattern and then explore the relationship between these pattern interfaces.

Schematic Diagram

The schematic diagram for this design model represents the pattern interfaces and the relationships between pattern interfaces. We start with the patterns in the Pattern-Level model, then reveal the interfaces of each pattern. As discussed in Chapter 4, pattern interfaces could be interface classes, interface operations, or both.

**Interface Classes**: An interface class is identified as one of the internal classes of the pattern. We recall that the interface class can be either a client or a server. The direction of the arrow in the relationship with other pattern interfaces illustrates the role that the interface plays.

**Interface Operations**: Each interface operation is identified as an operation in one of the pattern's internal classes. One internal class can implement one or more interface operations.

The schematic diagram for the Pattern-Level with Interfaces model represents pattern instances (with their identifiable pattern type), interface classes, interface operations, and relationships between interfaces. A representation of the model that captures these design rules is shown in Figure 5-4 (and later in UML notation).

**Figure 5-4. A schematic diagram for the Pattern-Level with Interfaces model.**

In Figure 5-4 the pattern instances are represented as bounding boxes (frames) with interfaces. Pattern instances are identifiable using their name and type. Interface classes are represented as tabs labeled with the class name adhering to the pattern frame. Interface operations are represented as rounded ellipses labeled with the Class Name::Operation Name. This diagram will be replaced with the UML model as discussed later in this chapter.

Relationships
The uses relationships in the Pattern-Level view should now be refined in terms of relationships between pattern interfaces. The following are the possible relationships between pattern interfaces:

**Class/Class**: This is the UML class relationship, which could be aggregation, association, or dependency.

**Class/Operation**: An interface class can invoke an operation in another pattern. This type of relationship is expressed when it is still ambiguous to the designer which operation in an interface class interacts with this interface operation. For example, the above figure, Class2 in PatternInstance2 invokes ClassX::Op1 in PatternInstance1.

**Operation/Operation**: This type of relationship models interactions, which usually reflect the designer perception of lower design details. Interactions are useful in developing behavioral description that could later be used to develop sequence diagrams.

**Design Decisions**

At this phase, the pattern interfaces are declared in the model. The designer is not concerned with internal design aspects of a pattern, since he uses pattern interfaces to glue patterns together. Patterns could have multiple interfaces according to the context in which they are used (see Chapter 4). The designer will select which interface to use according to the application at hand.

**UML Syntax Support**

In order to represent the Pattern-Level with Interfaces model in UML, we use the UML package and interface notations. UML packages are used to represent the pattern instance (UML package) and its type (UML stereotype). We use the notion of interfaces in UML to represent interface classes and interface operations. Figure 5-5 illustrates the Pattern-Level with Interfaces view using UML syntax.

**Figure 5-5. The Pattern-Level with Interfaces model using UML syntax.**
The UML interface notation (circles) is used to represent the interface classes. To represent interface operations, we use the UML interface notation (circles) where the name of the UML interface is the class name that implements the interface operation. The operation name is listed under the class name. Each interface is realized by a pattern; the UML realization relationship is used to tie an interface to its implementation (pattern).

Example

Let's consider the same example of the Reactor and Composite patterns shown in Figure 5-3. The Reactor pattern is a robust design for a system that receives events, manages a set of event handlers, and dispatches the event to the appropriate handler. The interfaces of the Reactor pattern are the abstract EventHandler class, which is the interface that all the concrete event handlers have to comply with, and the Reactor class, which is the class responsible for scheduling events and dispatching them to event handlers according to the type of the event.

The Composite pattern is a robust design for a system that provides a unique interface to a complex structure of objects that could be simple or composites. The interface to the composite structure is the Component class, which is considered the interface for the Composite pattern.

Therefore, the design in Figure 5-3 is further refined to represent the MyReactor and CompositeReactors relationship using the Pattern-Level with Interface diagram as shown in Figure 5-6.

Figure 5-6. An example of the Pattern-Level with Interfaces diagram in UML.
Detailed Pattern-Level Model

The Detailed Pattern-Level model builds on the Pattern-Level with Interfaces models to further take the design into a lower level of details. The purpose of this view is to explore the internal details of each pattern and identify the internal classes that implement the pattern interfaces. In order to develop these models, we need to reveal the internal structure of the patterns. Since we use constructional design patterns (as defined in Chapter 4), then each of the patterns that we use has an internal design structure in terms of a set of interrelated classes. We use this internal design structure in developing the Detailed Pattern-Level models.

Schematic Diagram

The schematic diagram for this design model represents the patterns and their internal structure. In this diagram the patterns are encapsulation mechanisms that hold a set of classes. The diagram represents the pattern instances and their type, interfaces, and internal class diagram. In addition, the relationship between the pattern interfaces and its internals is established. A representation of the model that captures these design rules is shown in Figure 5-7 (and later in UML notation).

Figure 5-7. A schematic diagram for the Detailed Pattern-Level model.

Relationships

A connectivity mechanism is used to show which elements of the pattern internals are exposed as interfaces. For instance, the schematic diagram in Figure 5-7 uses dotted lines to show the connections between the interfaces and the internals of a pattern. An interface class is connected to an internal class of the pattern—for example, Class1 in PatternInstance1. An interface operation is connected to the method of the class to which it belongs—for example, ClassX::Op1 in PatternInstance1.
Design Decisions

There are no explicit design decisions taken at this level; it is a refinement phase. If the Pattern-Level view is complex, it is preferable to express each pattern (or group of patterns) in a separate design sheet.

UML Syntax Support

This step is a refinement phase that uses the internal class diagram of the pattern. We use UML class diagrams to represent the internal design of each pattern. Therefore, the design in Figure 5-5 is further attributed with the UML class diagram of each pattern. To establish relationships between interfaces and internal classes, the name of the internal class that implements the interface is set to match the name of the interface. Figure 5-8 illustrates the Detailed Pattern-Level models using UML syntax.

Example

Referring back to our example, we will expose the internal structure of the Reactor and Composite patterns. The internal structure of a Reactor pattern can be found in D. Schmidt’s “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching” (1995b). It consists of the abstract EventHandler class, which is the interface that all the concrete event handlers have to comply with; the Reactor class, which is the class responsible for scheduling events and dispatching them to event handlers according to the type of the event; and the ConcreteEventHandler classes, which implement the EventHandler interface.

The internal structure of a Composite pattern can be found in the GoF book, Design Patterns: Elements of Object-Oriented Software (1995). It consists of the Component class, which is the interface for the structure; the Leaf class, which implements the Component interface but does not contain other objects of type Component; and the Composite class that implements the Component interface and
consists of other components that it manages. The Detailed Pattern-Level diagram for this example is expressed in UML syntax in Figure 5-9.

**Figure 5-9.** An example of the Detailed Pattern-Level diagram in UML.
Characteristics of the POAD Design Models

The three types of model diagrams that we discussed above are used for the structure composition of constructional design patterns in the POAD process. These models possess some useful characteristics: hierarchy, traceability, and composability.

Hierarchy

The three design models reflect different levels of abstraction. A high-level design decision is captured in the Pattern-Level models, where the selection of patterns and their relationships is made. At a lower level of abstraction, the Pattern-Level with Interfaces model represents the interfaces between patterns. Finally, the pattern internals are captured in the Detailed Pattern-Level model. This hierarchy property of the POAD composition models allows the internals of the pattern to be explicitly suppressed at one abstraction level and then expressed at lower design levels. This hierarchical feature is essential for complex applications. For these applications, patterns can be used as building blocks and we need not express their internal structure. We expose those parts of the pattern that ought to be glued together (interfaces). To further refine the design, we zoom into the pattern's internal design to view its internal details.

Traceability

A constructional design pattern is composed of several collaborating classes. From a composition perspective, the pattern is the container and its constituting classes represent the content. The interfaces provided by the pattern are implemented by its internal classes. A traceability mechanism should be provided to trace from high-level models in terms of patterns to lower level models in terms of its content. The three design models used in POAD facilitate the traceability from high abstraction level to lower levels. For example, pattern dependencies in Pattern-Level view are traceable to class/class, class/operations, and operation/operation relationships in the Pattern-Level with Interfaces view. The interfaces in Pattern-Level with Interfaces are traced to pattern internals in Detailed Pattern-Level view. This traceability feature enables the designer to navigate the models up and down the various abstraction layers.

Composability

Composability refers to the property of the models that enables model elements to be plugged together. The design artifacts in each pattern view are pluggable. At the highest level, patterns are glued together using pattern dependency relationships. At the Pattern-Level with Interfaces view, pattern interfaces allow patterns to be glued together using interface relationships, which are further traced into class diagrams. The ability to maintain the composition property at various levels is a good feature of the POAD approach.
Summary

In this chapter we defined the design models that we use for structure composition of patterns in the POAD approach. We used three hierarchical traceable views: Pattern-Level, Pattern-Level with Interface, and Detailed Pattern-Level diagrams. UML syntax is used to represent the model elements in each of these views. These models are used as an initial step in developing the class diagram of the application. In part IV, we use several case studies to illustrate how to use these models as part of the POAD development process.
Chapter 6. UML Support for Design Patterns

In Chapter 5 we discussed the design models used to support the POAD development process. The models that we use in the development process should be versatile; they should be commonly used and understood and should comply with some modeling standard in the field. POAD uses constructional design patterns that are strictly OO in nature. Hence, OO models should be used. Although it requires more than design models to capture the essence of a pattern, we focus in this chapter on the modeling aspects to support a design and development process. Other aspects of a pattern include usage tradeoffs, forces, consequences, and examples. These can be expressed by other means, such as in text or tables.

UML is the result of a unification effort of many OO modeling techniques. A development methodology in the OO paradigm should utilize existing UML models as much as possible. It might not be possible in all cases to use existing UML syntax and semantics to describe models for a particular development methodology. As a result, analysts and methodologists expend effort to integrate and extend UML with syntax and semantics that are required by a specific analysis and design methodology. There are two ways to extend UML modeling capabilities. The first, and also the easy way, is to use UML extension mechanisms. UML provides several extension mechanisms for various modeling constructs, such as stereotypes, constraints, tag definitions, and tagged values. The second, and also the hard way, is to add new modeling constructs to the language. This is usually done at the metamodel level where the concept that the new construct represents is related to other UML concepts as defined in the UML metamodel. Designers are always encouraged to use the first approach.

There are also situations in which designers use different UML constructs to represent the same design concept. For example, to model a design pattern in UML, designers utilize different UML constructs: class diagrams, packages, templates, stereotypes, and so on. This often happens because UML is developed to be a generic modeling language that is not tailored for a particular design methodology or process. Therefore, each design methodology utilizes the best UML constructs to highlight special aspects of its design process, approach, or concepts.

In this chapter we discuss UML support for modeling design patterns and explain approaches to model a pattern using UML. We also explore limitations in modeling constructional patterns using UML notation and semantics. Formal specification and metamodeling issues are deferred to Chapter 15. It is the objective of this chapter to discuss the relation between modeling design patterns and UML for the following purposes:

- when using design models to present a design pattern, we should be building on existing models that are the result of rigorous research and practice efforts. UML is a de facto standard for modeling OO applications, and hence POAD representation of patterns and their composition should be tightly related to UML modeling constructs.

- we should utilize UML modeling capabilities to the limit and identify any UML notation and semantics limitation in supporting the POAD development methodology.

- The models used in POAD should be related to the ongoing research of the Object Management Group (OMG) to standardize OO modeling languages.

- A development process should be supported by a development tool. Tools that support OO analysis and design processes adhere to the notation and metamodel specified in UML. We use UML to describe POAD models in order to encourage tool developers to integrate the development methodology in UML-supported tools.

- Using UML to describe the development process models, we assist methodologists and modelers in practicing and understanding POAD by relating its models to versatile UML models.

We begin by summarizing approaches to model patterns in UML and discuss the weaknesses and strengths of each approach. These approaches are then compared at the end of the chapter.
Patterns as Mechanisms

An OO design pattern defines the structure and collaboration between a group of classes to solve a particular design problem [Gamma et al. 1995]. With focus on the collaboration aspects, Booch, Rumbaugh, and Jacobson, in *The Unified Modeling Language User Guide* (1999), define a design pattern as a *mechanism* that applies to a society of classes. A mechanism is then modeled as a collaboration. Collaborations are useful in modeling *behavioral* patterns. A behavioral pattern mainly focuses on the interaction between its constituting classes. [1] Modeling patterns as collaborations capture the behavioral aspect of a pattern, that is, the interaction between its constituents. The interactions defined within the collaboration specify the communication between the instances of the pattern participants when they perform the behavior defined in the solution section of the pattern.

[1] Limitations of mechanisms and using role diagrams to model behavioral patterns are discussed in [Riehle 1996].

In UML the details of a collaboration can be captured in a collaboration diagram or an interaction diagram, which illustrate messages and invocations between objects instantiated from the pattern classes. Hence, a design model for a pattern will have a class diagram to capture its structure and an interaction diagram to capture its behavior. But these two diagrams capture the internal design of the pattern and have little to do with how the pattern will be modeled as a composition unit in the application design. For example, Figure 6.1 illustrates the structure (class diagram) and the behavior (interaction diagram) of the Observer pattern. These diagrams capture the internal design and behavior of an observer and do not represent the Observer pattern as a composition unit.

**Figure 6-1.** Class diagram (a) and interaction diagram (b) of the Observer pattern.

It is also possible to present a collaboration as a single entity that can be viewed from outside as a modeling construct. For example, this could be used to identify a design pattern in a system design. Figure 6-2 illustrates the UML construct used to model a collaboration.

**Figure 6-2.** Modeling a pattern using a UML collaboration construct.
In UML, a pattern is a synonym for a template collaboration that describes the structure of a design pattern. A parameterized (or sometimes called a template) collaboration is a way of modeling collaboration between participants where those participants are determined by input parameters. During instantiation of the UML parameterized collaboration construct, actual model elements are substituted for the parameters in the pattern definition. A parameterized collaboration in UML is represented as a dashed ellipse containing its name. A dashed line is drawn from the collaboration symbol to each of the symbols denoting the participants. Each line is labeled by the role played by the participant in the context of the collaboration. Therefore, a parameterized collaboration can show the use of a design pattern and the participants of the pattern.

For example, Figure 6-3 illustrates how the Observer pattern is modeled in UML using a parameterized collaboration. The dashed ellipse is the parameterized collaboration representation of the Observer pattern. The collaboration is parameterized with two parameters: the Subject class and the Observer class. Each of these two parameters is shown in the rectangular box attached to the collaboration for illustration (not specified in UML). When the designer uses this parameterized collaboration construct (the dashed ellipse) in the design, he has to attach concrete classes to the collaboration. Those classes play the roles defined by the collaboration parameters. For example, in the class diagram of Figure 6-3, the Sensor class is the Subject parameter to the collaboration, and the Controller class is the Observer parameter.

**Figure 6-3.** An Observer pattern modeled in a UML class diagram as a parameterized collaboration.

In some cases it is convenient to show the internals of a parameterized collaboration within the collaboration icon (the dashed ellipse). For example, the Sensor and Controller classes of Figure 6-3 can be shown inside the dashed ellipse collaboration construct, as shown in Figure 6-4.

**Figure 6-4.** Illustrating the internal design of the pattern inside the parameterized collaboration symbol.

Now we look at the possibility of using collaborations to model constructional design patterns as used in the POAD approach. A constructional pattern, as we discussed in Chapter 4, represents a structure of classes intended for use in constructing an application class diagram. The main focus is using the pattern structure in terms of class diagrams and not its behavior as modeled by collaboration. The approach of using collaborations to model a mechanism (design pattern) lacks the visual hierarchical view of a design. Collaborations and parameterized collaborations do not offer interface for composition; they are mainly used to identify participants that communicate with
Assume we have a class diagram with several classes that belong to multiple patterns. We will still have the large population of classes in one class view in addition to collaboration constructs that are added to pinpoint participants of a specific pattern. Hence, a true hierarchical view of the design is difficult to achieve. To get a hierarchical view we would rather provide a view of patterns as model artifacts—for example, packages, which encapsulate other design elements (class participants). These patterns could later be traced into lower design phases as class diagrams. Figure 6-5 is taken from the UML notation manual and illustrates that the use of collaboration might result in a cumbersome design that is not hierarchical and is difficult to comprehend.

**Figure 6-5. A class diagram of several patterns modeled as parameterized collaborations.**


---

**Takeaway 1**

Collaborations (or parameterized collaborations) are used to model mechanisms. They have some limitations in modeling constructional design patterns because they are not meant for modeling the structure of a design in terms of class diagrams. They do not provide interfaces for composition and hence do not support a hierarchical view of the application design.
Architectural Patterns

A framework is an architectural pattern that provides an extensible template for applications within a domain [Booch et al. 1999]. In this context, a framework is a large collection of classes that encapsulates multiple mechanisms. Frameworks are larger OO constructs than constructional design patterns. A framework is typically composed of several design patterns, classes, or templates.

Frameworks are larger modeling constructs than design patterns; they are used to model the entire application or a subsystem of the application. On the contrary, a constructional design pattern defines a smaller structure of design classes intended to solve smaller design problems, and hence granularity becomes a key difference when modeling frameworks and patterns. Granularity is an important issue for instantiation, extensibility, and composition purposes.

Instantiation mechanisms of a pattern or a framework in a domain-specific application are different. Instantiation of a design pattern is often a white box mechanism that requires the designer to have good understanding of the pattern internals. A framework is often instantiated through some extension points or hot spots. Hot spots should be minimal compared to the entire framework design; otherwise, the benefits from using the framework are not achieved. In many cases the designer might only need to know about the interfaces mandated by the framework for the application specific components that he is adding to the framework.

Composition with other modeling constructs is another difference between patterns and frameworks. Due to the difference in the number of classes encapsulated by each modeling construct, defining interfaces for either of them as well as composing with other modeling constructs could be different. Identifying interfaces of a framework are more difficult due to the larger number of classes encapsulated inside the framework. A framework interface could be a pattern by itself; for example, the Façade design pattern [Gamma et al. 1995] can be used as interfaces for subsystems and frameworks.

Extensibility is another difference between frameworks and constructional design patterns. Since frameworks have a larger population of classes, framework extension through inheritance or composition is possible. In many design patterns, extension of the pattern is feasible through inheritance.

Therefore, from a conceptual point of view, the essence of constructional design patterns should not be confused with specific frameworks. A constructional design pattern has fewer classes. Specific OO frameworks consist of a larger number of classes, and it is harder to define their interfaces. Moreover, frameworks themselves can be composed of a number of design patterns. Frameworks are modeled in UML as stereotyped packages. In UML “a framework is a stereotyped package that contains model elements which specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns, or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks” [UML 2002].

Takeaway 2

Constructional design patterns (as design artifacts) differ from OO frameworks (architecture patterns) in granularity, extensibility, and possibly interfaces. Hence, it is semantically incorrect to consider a constructional design pattern as a stereotyped framework. Moreover, a framework is not just one pattern; it could be a composition of multiple patterns and other classes that do not belong to a particular pattern.
Patterns as Packages

A package is a general-purpose UML modeling technique for organizing elements into groups. A package plays the role of a container that encapsulates its constituting elements. Since a package is a general-purpose grouping mechanism, it can contain other UML constructs such as classes, components, use cases, and other packages. Packages own model elements and can be used as the basis for configuration and access control. A UML model element can be owned by a single package.

To specialize the use of a package in a development process, the UML stereotype extension mechanism can be used. For example, a package can be used to model a framework by stereotyping the package with the framework keyword. A stereotype package is also used to model a subsystem where a subsystem is "a grouping of elements of which some constitute a specification of the behavior offered by the other contained elements" [Booch et al. 1999]. An example of a stereotyped package is shown in Figure 6-6. Using this UML notation, the package symbol can be used to represent a group of design elements that are encapsulated and hidden by the package. To further identify the elements contained by the package, another design diagram could be attached to the package representation; for example, a class diagram can be used to model the internals of the package in terms of classes and their associations. UML packages can be hierarchical; a package may contain other packages.

Figure 6-6. A stereotype package.

In POAD, constructional design patterns are used as building blocks of the application design. Therefore, packages are suitable constructs to be used in modeling patterns. They provide an encapsulation mechanism by which pattern details can be hidden. They provide a hierarchical view of a design because we can model the application as packages and zoom in to the content of each package. They also provide a visibility mechanism to control the visibility of the elements of the package where public elements are considered the package interfaces. Using a UML stereotyping mechanism with packages, we can represent a constructional design pattern as a package with a stereotype that reflects the pattern instance type. For example, a SensorObserver instance of the Observer pattern can be represented as shown in Figure 6-7. This technique was used in the POAD design models that we discussed in Chapter 5.

Figure 6-7. A stereotyped package that models a pattern.
In UML, packages are usually modeled as containers for classes; therefore, they appear in package and class diagrams. Figure 6-8 illustrates an example of a package diagram.

**Figure 6-8. A package diagram.**


The semantics of the package diagrams as defined in UML have some limitations when applied to modeling composition of patterns:

1. **Relationships.** In UML, the relationship between two packages is either dependency or generalization. The dependency relationship is semantically an import or export; that is, one package has access to the contents of another (such as Java and Ada packages). The generalization relationship is used to specify families of packages. The relationship between constructional design patterns is different. We neither want to import parts of a pattern into another nor generalize two patterns and construct a generic pattern. A pattern is not usually a part of another pattern [Gamma et al. 1995][2]

[2] Another opinion is given by Riehle (1997), who sees that we can create composite patterns out of simple ones.

The relationship between constructional design patterns in POAD is a dependency relationship that is semantically a uses relationship and is not an import or access relationship. To overcome this semantic issue, we can stereotype the relationships to be uses rather than generalization or dependencies. The models developed in Chapter 5 assume that the uses stereotyping is used.

2. **Interfaces.** Since a package is a general-purpose grouping mechanism, its interface differs according to the nature of the encapsulated elements. UML defines an interface as a modeling artifact that is a collection of operations. When using a package to model a pattern, the pattern internals are classes; thus interfaces could be the classes themselves or operations of interface classes. A package uses a visibility mechanism (+ or – symbols) to define interfaces; this is because the package owns its elements and its elements are not part of other packages. However, when using packages to model pattern compositions in POAD, we use the UML interface construct to model an interface to a package, as discussed in Chapter 5. For example, a CompositeReactor instance of type Composite pattern will be represented as a package with Composite stereotype and the Component class as the UML interface, as illustrated in Figure 6-9.

**Figure 6-9. A CompositeReactor instance of type Composite pattern.**
Packages are general-purpose grouping mechanisms in UML. They are the closest UML solution to model constructional design patterns. Using packages to model constructional design patterns requires additional stereotyping to reflect the pattern type and relationship between patterns.
Patterns and Components

In POAD we use patterns as design building blocks that encapsulate the structure of a set of collaborating classes. In this context the designer might consider using UML components and component diagrams to model constructional design patterns and their composition. However, physical components and design components are different, as discussed in Chapter 4. There are conceptual differences between components (as specified in UML) and constructional design patterns (as design components in the POAD approach).

In UML a component "represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces" [UML 2002]. Components are physical packages used to model source code, executables, or physical databases. Constructional design patterns are logical packages composed of a set of classes at the design phase, which don't mandate a separate or single physical packaging. Logical packaging is used to organize design elements, while physical packaging is used to define implementation deliverables.

Another difference between components and patterns is the composition and traceability semantics. A component as a physical unit encapsulates the implementation of other modeling artifacts, such as classes, interfaces, and collaborations. A component can contain other components. UML dependency relationships are used to illustrate containment dependency, which shows how a component contains other components. We can also use dependency relationships to illustrate usage dependencies on other components. Constructional design patterns are logical packages that visually encapsulate their constituting classes but do not encapsulate their implementation. Component diagrams are implementation diagrams, while pattern diagrams are design diagrams.

Takeaway 4

Constructional design patterns (logical packages) cannot be modeled using UML components (physical packages) because they are conceptually different encapsulation and containment mechanisms. Hence it is semantically incorrect to use component diagrams to model pattern composition.
Modeling Pattern-Oriented Designs

The POAD approach that we use to build pattern-oriented designs using constructional design patterns as their constituting building blocks should be supported by logical model views that have the following properties:

- **Hierarchical (multilevel abstraction).** Hierarchy is the ability to decompose the application into large-grained logical modeling artifacts at a high level of abstraction and further decompose large-grained artifacts to finer grains at lower abstraction levels. In UML, several modeling constructs, such as packages and components, can be used to provide hierarchy. Packages provide a good containment mechanism to support hierarchical design models.

- **Traceable.** Traceability is a modeling property that supports uninterruptible transition from high-level abstractions to lower levels. The effort in tracing high-level abstractions to lower levels should be minimized both semantically and visually. UML package diagrams provide support for tracing the package from a package diagram into the internal structure of the package in terms of a class diagram that represents the package structure.

- **Composable.** For each level of abstraction, interfaces should be well defined and should facilitate the integration of the building blocks of each abstraction level.

The models that we use (discussed in Chapter 5) provide support for these three properties through the use of UML packages, interfaces, dependencies, and stereotyping extension capabilities.
A Comparison

UML offers several modeling constructs that we can use to model a design pattern. There is no absolute correct or incorrect construct to use in modeling a pattern; the selection of the UML construct to use depends on how the design methodology uses patterns. Therefore, the designer and user should expect to see different development processes use different UML constructs to model patterns and pattern compositions.

To compare different UML constructs, we first define the comparison criteria. The following is a list of criteria that we use for comparison:

- **Structural/behavioral modeling.** This criterion defines which aspects of the pattern we are interested in. The structural models capture the skeleton of the design, while the behavioral models capture the interactions and the dynamics.

- **Hierarchical.** Hierarchical constructs are used to provide a coarse-grained view of the design at one level and a fine-grained view at lower design levels.

- **Traceability.** Models at one level of abstraction should be traceable to lower abstraction levels. For instance, a representation of the pattern as a composition of classes could be viewed as a design component at one level that is further traced to the pattern internals in terms of classes and objects at lower levels.

- **Semantic relationships.** When using a UML construct to model a pattern, we have to consider what types of relationships are supported in UML between instances of that modeling construct.

- **Physical/logical.** A UML modeling construct can be a physical component that is supported as an implementation unit or a logical component that is used in the modeling space only and has no exact mapping in real implementation.

**Table 6-1** summarizes the comparison. From this table we find that components cannot be used to model constructional design patterns because patterns are not physical, deliverable implementation units. We also find that mechanisms (such as collaborations and parameterized collaborations) are not perfectly suitable for POAD because they are more concerned with the behavioral aspects of the pattern than with the structural aspects. Packages seem to be the most suitable UML construct to use after using some stereotyping mechanisms to extend package modeling to suit modeling of constructional design patterns and their composition.

**Table 6-1. Comparison between UML constructs to Support Constructional Design Patterns**

<table>
<thead>
<tr>
<th>Constructs</th>
<th>Mechanisms</th>
<th>Packages</th>
<th>Components</th>
<th>Constructional Design Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural/behavioral</td>
<td>Behavioral</td>
<td>Structural</td>
<td>Structural</td>
<td>Structural</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Traceable to</td>
<td>Group Of Objects</td>
<td>Group Of Classes</td>
<td>Physical implementation</td>
<td>Group Of Classes</td>
</tr>
<tr>
<td>Support for Interfaces</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Connectivity Between Interfaces</td>
<td>No</td>
<td>Uses +/- Symbols; Cannot Be Connected</td>
<td>Connects interfaces to one another</td>
<td>Connects individual interfaces to one another</td>
</tr>
<tr>
<td>Semantic Relationships</td>
<td>No</td>
<td>Access Generalization</td>
<td>Uses</td>
<td>Uses</td>
</tr>
<tr>
<td>Logical/Physical</td>
<td>Logical</td>
<td>Logical</td>
<td>Physical</td>
<td>Logical</td>
</tr>
</tbody>
</table>
Summary

In this chapter we discussed UML support for modeling design patterns. We illustrated how various UML model constructs can be used to model a pattern and the capabilities provided by each modeling construct and its limitations. We also discussed whether each technique is suitable to model constructional design patterns and described how the use of stereotyped packages suits the problem of describing patterns as design components.
Part III: Process Aspects of POAD

In Part III we discuss the process aspects of POAD. Chapter 7 describes the procedures and steps to apply POAD in the design of software systems. We first discuss the stringing and overlapping pattern composition approaches, then illustrate how POAD reaps the benefits of the two worlds. This chapter also summarizes the analysis, design, and detailed design phases of POAD and illustrates the overall outline of the process. Chapters 8, 9, and 10 elaborate on the analysis, design, and design-refinement phases, respectively, and discuss the development and modeling steps within each phase.
Chapter 7. POAD: The Process

In Part II we discuss the technological aspects of POAD in terms of the models used to develop an application design and how these models utilize UML constructs. Yet another set of models is not what POAD is about. Leaving the designer with only a set of models can sometimes be confusing. It is important to present an example of a process that can be used in developing these models; this is what we call the process aspects of POAD. Part III deals with applying POAD and utilizing the models discussed in Part II. This process is not the only process in which the designer can use patterns to develop software designs, it is the process that we find useful in developing the case studies discussed in the next part of the book. We intend to illustrate the development phases and the various steps within each phase. This is just a recipe of the analysis and design workflow that can be adapted accordingly.

The purpose of this chapter is to outline the POAD process and to illustrate how patterns can be utilized as design building blocks (design components). We first discuss two viewpoints about gluing patterns together at the design level and then describe how POAD evolves as the integration of the two viewpoints. The outline of the process shows how to glue the design structure of patterns at various levels of abstraction for the purpose of developing pattern-oriented designs.
Stringing Versus Overlapping

It is possible to make buildings by stringing together patterns in a rather loose way. A building made like this is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound [Alexander et al. 1977].

In the field of civil engineering, Alexander and colleagues, in their book *A Pattern Language*, discuss composing patterns based on their experience in making buildings. They discuss two techniques: stringing patterns together and overlapping them. Many of these principles apply to the design of software systems as well. Inspired by Alexander's approaches to make buildings, consider the two approaches to build software applications using design patterns:

1. **Stringing patterns.** In this design approach patterns are glued together to compose an application design. The glue here could be UML relationships between patterns as packages (for example, dependency between packages) or UML relationships between participants of the patterns (for example, UML association and dependency between classes of one pattern and classes of another pattern). The design is a loose assembly of patterns because it is made by simply stringing patterns and using all the internal participants of a pattern as independent design constructs. For example, suppose you have decided to use a Strategy pattern and an Observer pattern [Gamma et al. 1995]. If you use the stringing patterns approach, you will end up with a design that has all the classes of the Strategy pattern (Control, abstract Strategy, and ConcreteStrategy classes) and all the classes of the Observer pattern (abstract Observer, abstract Subject, ConcreteObservers, and ConcreteSubjects classes). This design is neither dense nor profound. It is not dense because you end up with a design that has a large population of classes. It is not profound because many classes have trivial responsibilities. The reason is that the design of many of these patterns has several classes that are only responsible for forwarding to other classes, acting as an interface to the internal design of the pattern, or representing a class that is intended to be part of the external system design, not the internal design of the pattern (i.e., a client class of a pattern).

2. **Overlapping patterns.** This approach advocates that many patterns should overlap in the same logical design. Overlapping means that a class as a participant in one pattern could be at the same time a participant of another pattern in the same application design. For instance, consider the above example of gluing together the Strategy and the Observer patterns: overlapping these two patterns could mean that the abstract Strategy class of the Strategy pattern plays the role of abstract Subject class in the Observer pattern. So, the designer will use one application class to play the role of both participants. As a result, the design is dense and it becomes more profound. It is dense because you will end up having fewer classes in the application design than the total number of classes in the patterns used to develop that design. It is profound because each class carries out several responsibilities.

With the overlapping patterns approach, we gain the advantage of having a fewer number of classes in the application design than the one produced by stringing patterns. However, there is one salient disadvantage. The pattern boundary is lost, so patterns are hard to trace.

With stringing patterns, we can always identify the pattern by circling the classes that implement it. When circling the classes of a pattern in the overlapping pattern design, we end up with so many intersecting circles.

To make things more concrete, consider an application in which the designer has decided to use the Reactor pattern [Schmidt 1995b] and the Composite pattern [Gamma et al. 1995]. This example is part of the case study design discussed in Chapter 12. Let us use these two patterns to illustrate the difference between the overlapping and stringing approaches. The class diagram model for each of the two patterns is shown in Figure 7-1.

![Figure 7-1. The class diagram for the Reactor pattern (left) and the Composite pattern (right).](image-url)
The Reactor pattern is a robust design for a system that receives events, manages a set of event handlers, and dispatches the event to the appropriate handler. It consists of the abstract `EventHandler` class, which is the interface that all the concrete event handlers have to comply with; the `Reactor` class, which is the class responsible for scheduling events and dispatching them to event handlers according to the type of the event; and the `ConcreteEventHandler` classes, which implement the `EventHandler` interface.

The Composite pattern is a robust design for a system that provides a unique interface to a complex structure of objects that could be simple or composite. It is composed of the `Component` class, which is the interface for the structure; the `Leaf` class, which implements the `Component` interface but does not contain other objects of type `Component`; and the `Composite` class, which implements the `Component` interface and consists of other components that it manages.

Suppose you want to implement a reactive system. You choose to use the Reactor pattern, but then you find that the handlers for the application-specific events are not simple objects; instead, they could be complex objects containing other objects that react as well to the events. So, you decide to use a Composite pattern. Now, you have two patterns to glue, the Reactor and the Composite patterns.

The first solution is to string the two patterns together by establishing a relationship between the `Component` class of the Composite pattern and the `EventHandler` class of the Reactor pattern. By stringing the two patterns, we end up with the design shown in Figure 7-2, which contains all the classes of the two patterns. What is not profound about that design is that the handlers are in reality the composite components themselves rather than handlers that use or reference composite components.

Figure 7-2. Stringing the Reactor and Composite patterns.

From the stringing model class diagram, we can generate Java source code using any UML modeling tool that supports code generation. The following is the snippet of code used for a concrete event handler class:

```java
public class concreteHandler1 implements EventHandler {
    protected Component myComponent;

    // operations
    public void handleEvent(Apple aEvent) {
        myComponent.operation();
    }

    public AppleHandle getHandle() {
        // application code goes here
    } // end getHandle
} // end concreteHandler1
```
As illustrated from the code in italics, the function to be handled by this concrete handler is now delegated to a component that could be either simple or composite. Then, the implementation of the operation() method of the composite pattern classes will do the application specific function. The role of the concrete handlers in this case has become just a simple forwarding mechanism. Such implementation is neither dense nor profound and will have many classes with trivial forwarding responsibilities like the concrete handler above.

The second solution is to overlap the two patterns. In this case, you know that your handlers are the composite objects. Thus, you decide that the EventHandler of the Reactor pattern and the Component class of the Composite pattern can be overlapped, and both roles are integrated in one class—call it EventHandlerComponent. This class will have the methods in both classes. Consequently, the concrete event handlers become concrete classes derived from the EventHandlerComponent class. The overlapped pattern design is shown in Figure 7-3.

![Figure 7-3. Overlapping the Reactor and Composite patterns.](image)

From the overlapping model class diagram, we can generate Java source code using any UML modeling tool that supports code generation. The following is the snippet of code for a concrete event handler class, which in this case plays the role of a leaf class in the Composite pattern and a concrete event handler in the Reactor pattern.

```java
public class LeafEventHandler implements EventHandlerComponent {

    // operations

    // handleEvent() method. This method is both:
    // The operation() method of class Leaf in Composite pattern
    // and
    // The handleEvent() method of class concrete
    // event handler in Reactor pattern

    public void handleEvent(ApplEvent anEvent) {
        // directly write the application code for handling this event here
    }

    public ApplHandle getHandle() {
        // application code goes here
    }

```

// other methods implementation goes here
}

// end concreteHandler1
As illustrated from the code, the operation `handleEvent()` represents the `operation()` method of class `Leaf` in the Composite pattern and `handleEvent()` method of class `concrete event handler` in the Reactor pattern. The class `LeafEventHandler` is both a `Leaf` class of the Composite pattern and a `concrete event handler` class of the Reactor pattern. From these overlaps comes the density and profoundness of this design.

*Are these two approaches independent? Must we construct a design that is either a sparse assembly or a condensed overlap of patterns? Can we use both?*

Recall the pattern composition approaches discussed in Chapter 3. Whether composing the behavior or the structure of the pattern, these approaches mostly fall into the second category, overlapping patterns. The overlapping is done by overlapping the role or behavior of the participants from each pattern or by overlapping the class participants of each pattern.

Clearly, the first approach, assembling and stringing patterns, is avoided by many designers. This can be attributed to the perceivable disadvantages of simply assembling patterns to produce designs. It is, however, the easier approach to practice. The stringing patterns approach provides good traceability from high-level designs, in terms of patterns, to lower-level designs, in terms of classes; simply group the classes of a pattern in one package or a template package [D'Souza & Wills 1998] that becomes the high-level view and use the pattern classes in the class diagram model that becomes the low-level design. In this case, you are not making any more design decisions than choosing the patterns and there is a one-to-one mapping between the class participants of each pattern and application design class. Moreover, this is currently supported by many design tools that use existing UML models.

The POAD approach reaps benefits from both the stringing and overlapping patterns worlds. It makes use of the simplicity and traceability of the stringing patterns approach and the density and profoundness of the overlapping patterns approach. In POAD the two approaches are not independent, and in fact they could be integrated in one process. POAD starts by assembling patterns at a higher level of abstraction using the stringing approach, provides models to trace the patterns to lower levels of abstraction, and then allows the designer to integrate lower-level classes to produce dense and profound designs.

We start by outlining the various phases of the process. We elaborate on each of those phases in the following chapters. The following section presents an overview of the process. Later we describe characteristics of the process and its supporting models. We then describe the application of the process to develop pattern-oriented frameworks and conclude the chapter with a discussion of what we perceive as advantages and disadvantages of POAD.
POAD Process Outline (The Nutshell)

The process aspects of POAD explains the phases and steps to develop an application design using patterns. The output of conducting the process is a pattern-oriented design or a pattern-oriented framework. The main input-resource required by the process is a library of constructional design patterns. We use the term phase to refer to the well-known software development phases: analysis, design, detailed design, implementation, testing, and so on. We use the term step to refer to an activity or a process step that the analyst or designer conducts within a particular phase. We explain each step of the development phase by purpose, process, and product: the purpose section explains why a designer would conduct that step; the process section describes the activity that the designer performs in that step; the product section describes the expected output of that step.

In general, the POAD process has three phases: an analysis phase in which a set of patterns are selected from a domain-specific library; a high-level design phase in which patterns are glued together using pattern composition models to produce an initial class diagram; and a design refinement phase in which the initial class diagram is processed to produce a more dense and profound class diagram for the application.

The waterfall linear model we use to explain the POAD process is meant to give an overview of the phases and steps within each phase. Incremental and iterative development is encouraged and in fact is illustrated using some loops in the process. In practice, incremental development is better than trying to do it all up front. Iteration from one step to earlier steps is encouraged and is made possible by preserving design models that are produced by each step and providing traceability across these models using a development environment or a tool support.

Figure 7-4 illustrates the legend that we use to describe the POAD lifecycle. We use square boxes to represent an artifact (product) and ellipses to represent a process (activity) in a certain step. Figure 7-5 illustrates the overall development phases of POAD. Figure 7-5(a) shows the analysis, design, and design refinement phases, each of which is explained in more detail in Figure 7-5 (b,c,d).

Figure 7-4. Legend used to describe POAD.

Figure 7-5. The POAD process: (a) overall phases, (b) analysis, (c) design, and (d) design refinement.
In Figure 7-5(b) the analysis phase consists of a requirements analysis step followed by a selection step. In requirements analysis, conceptual or logical components are identified, and in the selection step a set of patterns needed for the logical components is selected. The design phase pattern diagrams are constructed as shown in Figure 7-5(c). The design refinement phase produces an optimized class diagram, as shown in Figure 7-5(d).

In the following subsections, we briefly describe the steps within each of the three phases. The following three chapters elaborate on the details of each phase and provide simple, illustrative examples. Applications of this process are illustrated in the case studies in Part IV.

Analysis Phase

Purpose

The purpose of this phase is to analyze the application requirements and decide on the set of design patterns that will be used in designing the system.
**Process**

Using the application requirements as an input, the analyst conducts several steps in order to decide on a set of design patterns to retrieve from a library of application specific-patterns. In this step, UML use case diagrams and sequence diagrams can be used to identify the patterns needed to support these interactions. Those patterns will be used in building the system design. The analyst is not concerned at this level with the internal details of the patterns. For instance, it is not needed at this phase to learn how the pattern solves the problems posted in the application requirements; instead, the analyst is concerned with determining whether or not the pattern can be used and why it is better to use it than to use other patterns in the library or a proprietary solution. The analyst first identifies conceptual packages or components for which he can delegate the system functional properties. These functional responsibilities define the overall role of a conceptual component and are used to guide the process of pattern selection by identifying the necessary responsibilities required from a particular pattern to implement this conceptual component.

As part of the analysis phase, the analyst searches pattern catalogues for candidate patterns. To make the search more effective, a domain-specific library of patterns rather than a general-purpose library should be used. Similar to traditional software component library management, this process includes acquaintance and retrieval.

A blind definition of conceptual components will not work in practice. This is because it is difficult to find a constructional design pattern (or a set of patterns) that implements the functionality of the defined conceptual component. Therefore, this step usually relies on the acquaintance of the analyst with the existing library of patterns from which the selection is made. In fact, some of the patterns in the library may motivate the analyst to identify conceptual components that are easily designed and implemented using these patterns. For instance, a pattern language of state machines and statecharts motivates the analyst to develop a model (or an architecture) that is based on collaborating components whose behaviors are substantially modeled in statechart specifications. Selic, Gulleksen, and Ward's *Real-Time Object Oriented Modeling* (1994) has architecture models based on such techniques. Acquaintance with the library content is an activity that the analyst will consider in the analysis phase.

The retrieval process defines how to select a pattern from a catalogue. Retrieval of software assets from libraries has been a research topic for decades. Several techniques have been proposed, and several issues have been identified, including matching criteria, exact and approximate retrieval, and assessment criteria such as precision, recall, coverage ratio, complexity, and automation [Mili et al. 1998]. Design patterns are no exceptions. Moreover, retrieval of patterns from databases could be easier because the pattern community has done an excellent job sticking to the definition of patterns using one of few templates, which makes it easier for automated retrieval by matching the problem or the intent section of a pattern with the problem to be solved.

**Product**

The product of this phase is a set of design patterns chosen by the application analyst. We note that at this phase nothing about the details of the pattern is declared; that is, neither the class model nor the interaction (behavioral model) are involved in the analysis step.

**Design Phase**

**Purpose**

The purpose of this design phase is to develop the application design by composing the patterns that have been selected in the analysis phase. As part of the design phase, the activities within this phase will produce design models at various levels of abstraction. These models should be easily traceable from higher levels of abstraction to lower abstraction levels, and vice versa.
Process

First, the designer creates instances of the patterns that were selected in the analysis phase and identifies the relationships between these instances. This is the direct application of the stringing pattern approach that results in a Pattern-Level logical view (as discussed in Chapter 5). This activity includes several subactivities such as creating pattern instances by turning a design pattern into a tangible design artifact; defining relationships between instances of patterns; and constructing Pattern-Level diagrams for the overall system and for each individual subsystem.\[2\]

Breaking down a system into subsystems is encouraged for large, complex systems such as those discussed in Chapters 13 and 14.

The designer proceeds from the Pattern-Level diagram to create a Pattern-Level with Interfaces diagram. The designer analyzes the relationships between pattern instances and further traces these relationships to lower-level design relationships between pattern interfaces. He then identifies pattern interfaces and decomposes pattern dependency relationships to lower-level relationships between pattern interfaces. As a result, the Pattern-Level with Interfaces diagrams (as defined in Chapter 5), which are the refinement of the Pattern-Level diagrams, are produced.

Starting with the Pattern-Level with Interface diagram, the designer identifies details of the pattern in terms of class diagrams for the purpose of creating an initial class diagram for the system. As a result a Detailed Pattern-Level diagram is produced.

Product

The product of this phase is Detailed Pattern-Level diagrams, which are design models showing the design of the application as a set of instantiated patterns.

Design Refinement Phase

Purpose

The purpose of this step is to develop the profound, dense class diagram for the application, which will be given for developers to implement.

Process

The designer starts with the outcome of the design phase, the Detailed Pattern-Level diagram. He instantiates each pattern in the context of the application under development. This includes choosing names for pattern participants that are meaningful in the application context and defining application-specific names for operations in the pattern classes. These activities can be conceptually categorized as concretization, specificity, scoping, and revision of the design. These activities are discussed further by Keller and Schauer in “Design Components: Towards Software Composition at the Design Level” (1998) and will be discussed further in Chapter 10. As a result, an application-specific Detailed Pattern-Level design is produced.
Using the Detailed Pattern-Level diagrams, the designer develops an initial class diagram of the application design. To develop a class diagram from the Detailed Pattern-Level diagrams, the designer must trace all relationships between patterns at the domain-specific Detailed Pattern-Level to class relationships, which are associations, dependencies, generalizations, and so on. The resulting design is viewed as an initial UML class diagram, which is neither dense nor profound.

The class diagram obtained from gluing patterns together at the high-level design is neither dense nor profound. It has many replicated abstract classes of the same purpose due to using the same pattern in multiple instances. It also has many classes with trivial responsibilities. The designer uses reduction, merging, and grouping activities to optimize the design diagrams. Reduction is responsible for removing replicated abstract classes that result from using the same pattern in several instantiations in the same design—for instance, using two application-specific instances such as a FeedbackObserver and a SensorObserver pattern of type Observer pattern. These activities are further discussed in Chapter 10.

Starting with the refined class diagram and the pattern-oriented views of the design, the designer can further analyze the system using traditional OO design models such as collaboration, interaction, and statechart models.

**Product**

The product of this phase is an optimized class diagram for the application.
POAD and Code Generation

The POAD process is mainly concerned with the analysis and the design process. It does not address specific coding styles or which OO language to use in the implementation phases. POAD focuses on creating the design models from which detailed designs and implementation can be generated. As you may have noticed, the output from the process is simply class diagrams, which contain traditional OO models in terms of classes, associations, inheritance, and so on. Hence, the models produced from POAD can be implemented using many OO programming languages.

Now comes the question of what happens when the developer changes things at the code level. This is a common problem in any modeling environment: how to bridge the gap between coding and analysis and design models. Many UML modeling tools support the round-trip engineering principles in which the code is kept in synchronization with the models used to generate them. For instance, all the class diagrams can be kept in synchronization with the code modified by the developer, and all code can be regenerated taking into consideration portions of the code that are handled by the modeling tool and portions that are left for the developer to fill in.

With these engineering and reverse-engineering mechanisms in place, we can fairly say that code is kept in synchronization with the low-level design models such as the class diagrams. Now the questions is, how would this affect the models created by POAD in the analysis and design level? In the following chapters describing the detailed process and the case studies, we will find that POAD has its own traceability mechanisms that enable tracing from lower-level design models (such as class diagrams) to higher-level models (pattern-level diagrams, pattern interfaces diagrams, etc.), and vice versa. These traceability mechanisms can be considered the round-trip engineering at the analysis and design level and are provided automatically by tools that support the POAD methodology. With round-trip engineering mechanisms in POAD to bridge pattern analysis and design models with class diagram models, and round-trip engineering mechanisms between class diagrams and code as supported by current UML modeling tools, we have a complete synchronization between code and higher-level design models, as illustrated in Figure 7-6.

![Figure 7-6. Synchronizing code with models.](image-url)
POAD Characteristics

By observing the models in Chapter 5 and the overall outlines of the process described in the previous section, we can deduce some characteristics of the POAD approach. In this section we summarize some of these characteristics.

Pattern-Driven

Most OO analysis and design methodologies utilize classes and objects as design building blocks. Many OO modeling languages, such as UML, offer grouping mechanisms, such as packages and subsystems, to construct a larger-grain view of a design. These grouping mechanisms are used to manage the model complexity by viewing the design as a number of collaborating packages that are later decomposed and refined.

The POAD approach is pattern-driven, meaning that everything starts from instantiating patterns in the design. In POAD you build applications from large-grain design building blocks—that is, from constructional design patterns. In this context patterns are used as grouping mechanisms and as reusable design solutions at the same time. As a grouping mechanism, a pattern encapsulates a set of classes that solve a specific design problem. As a reusable design solution, a pattern provides an abstracted solution to common recurring design problems. POAD is based on patterns as reusable building blocks of application designs.

Component-Based Development

Component-based software engineering is emerging as a beneficial paradigm for software development that delivers on some of the promises of software reuse. The nature of a component determines when the component can be used in software development and at which development phases. A component can abstract a function, data, package, or system structure. In general, many people refer to a component as either a class, a fragment of code, a fragment of design, an executable module, a runtime link library (dynamic-link library, or DLL), or a static library. Components can be classified based on their nature as

- **Specification Components.** Specifying the expected functionality and behavior of a component gives the developer the freedom to implement the component in a variety of programming languages. An example of using specifications as components is illustrated by Iglesis and Justo in "Building System Requirements with Specification Components" (1998).

- **Executables.** Executable components can be static libraries, DLLs, or executable pieces of an application. Many literatures refer to components of this nature. Usually, the source code of these components is not available, but the guidelines for integrating them in development are given in the documentation distributed with the component.

- **Design Components.** A component can be a design principal or structure. Constructional design patterns, as discussed in Chapter 4, are used as design building blocks in constructing OO applications. Usually, those types of components are used as white boxes at the design level.

In POAD, applications are built using design components as their building blocks. This produces a design that is component-based in nature, but we have to remember that the components used here are white box design components. This does not lessen the importance of using interfaces to glue these components. In POAD, constructional design patterns are used as design components, and they possess interfaces, as discussed in Chapter 4.

Architectural Development
Software architecture considers the structure of an application rather than its function [Shaw & Garlan 1996]. Choosing a good structure affects the functional and nonfunctional properties of the application, such as ease of integration [Shaw 1995], performance, and robustness to changes [Dyson & Anderson 1997]. When developing an application architecture, we are not concerned with the implementation details. We are more concerned with allocating functionalities to components and defining their interactions.

In POAD we build applications from a set of constructional design patterns—that is, design components. We choose a pattern that solves a particular design problem and define the relationship between patterns and how they interact. Therefore, the models resulting from POAD can be used as architectural views of the application. The high-level decomposition of the applications is considered an architectural-based approach. In Chapter 5, we discussed the architectural aspects of the Pattern-Level view.

Library-Driven Development

POAD uses existing catalogues of design patterns. The approach heavily relies on the existence of reusable libraries of design patterns that can be browsed and queried. Therefore, many of the traditional reuse library issues concerning asset retrieval [Mili et al. 1998] still hold for pattern libraries. The nature of a pattern motivates a different approach to maintaining and browsing libraries of patterns as opposed to libraries of code assets. A library of patterns is essential for the POAD approach; however, issues related to constructing and maintaining pattern libraries are research topics of their own that require further study.

Design Reuse

The POAD approach encourages reuse at the design level by motivating the reuse of constructional design patterns to build application designs. As compared to code reuse, we believe that leveraging reuse to the design level has several benefits:

- Software design is the product of tedious and perplexing activities of software development, specifically the requirement analysis and application design. Design decisions are harder and more critical than low-level implementation decisions. Reusing successful design decisions improves the quality of software and decreases development risks.

- Design is an iterative process. Design can be guided by high-level analysis (top-down) as well as by low-level development (bottom-up) approaches. The top-down aspect ensures that the design complies with the user requirements, while the bottom-up aspect ensures that the design is implementable.

- Instances of reuse are more likely for designs than they are for executable or code assets. It is usually difficult to match component specifications or user requirements to executable component specifications. Matching to designs could be more feasible due to the possibility of white box reuse of designs through adaptation.

- Constructional design patterns are generic design solutions. Application designers make use of the common design solution and focus on application specifics.

- Programming languages do not provide the high-level abstraction desperately needed by system designers [Gil & Lorenz 1998]. Constructional design patterns provide this high-level abstraction by enclosing the parts that provide the meaning and functionality of the pattern.

Since POAD starts off with reusable design components, it encourages reuse at the design level.

Hierarchical Development

Hierarchical decomposition and aggregation is often referred to as a consist-of and part-of relationship. The models that we discuss in
Chapter 5 for POAD permit hierarchical decomposition. Considering the relationship between the whole and its parts, we can identify three types of hierarchical decomposition:

- **Encapsulation.** The aggregate whole hides the internal parts from the outside world.
- **Embedding.** The aggregate whole is composed of parts that could be visible to other artifacts in the outside world.
- **Virtual Decomposition.** The aggregate whole is just a concept and doesn’t exist as an artifact.

Using constructional design patterns as design components is a virtual encapsulation approach to decomposition. This is because the pattern itself may not be tangible as compared to classes that are implemented in code constructs and instantiate as objects. Thus a pattern may be considered as a virtual group. At the Pattern-Level diagrams, patterns encapsulate the details of their internals and offer interfaces.

**Iterative Development**

The development process is iterative. As analysis proceeds, using sequence diagrams, for example, the relationships between classes could change. These changes have to be reflected in the Pattern-Level diagrams. The tool supporting the POAD process should maintain the consistency between design models. The further detailed design steps follow traditional OO techniques by defining details and behavioral aspects of classes and other design artifacts.

[3] We further elaborate on tool support for POAD in Chapter 16.
Pattern-Oriented Frameworks

One particularly important use of POAD is in the development of frameworks. In Chapter 2, we discussed the role of design patterns in developing software frameworks and how other researchers and practitioners have found patterns useful in the development of robust maintainable designs. In this section we illustrate how POAD emphasizes this role.

In POAD, design patterns are used as building blocks of an application design. This could be the design of a single application or the design of a framework that will be instantiated in several applications. The class diagram model of a pattern provides several abstractions that are turned into concrete designs when instantiating the pattern in an application. When combining patterns together, we disregard the design details of each individual pattern, and we use them as abstract design constructs to develop frameworks. Hence, abstractions that are modeled in a design pattern are conveyed to abstractions in the design of the framework. POAD models the application at various levels of abstraction: you get higher levels of abstraction as composition of patterns and the lowest level as class diagram models. The Pattern-Level model is of particular interest in designing a framework because it is a documentation of the framework as a composition of patterns. Moreover, the framework designer will still provide the class diagram model of the framework. As a result, the framework user is provided both the high-level view as a composition of patterns and the low-level view as a composition of classes.

When instantiating the framework, the framework user is left with two choices: starting from the Pattern-Level model or from the class diagram model.

Pattern-Level Instantiation

Pattern-oriented frameworks can be represented in POAD at the Pattern-Level diagram as a composition of building boxes (blocks), that is, patterns. The framework user can initially use these diagrams as an architectural description of the system. The framework user instantiates the patterns by expanding them into classes, and then carries out the design refinement steps. The design is then refined and details are added. It is more comprehensive for the framework user to instantiate the framework starting at the Pattern-Level because design details are hidden. The disadvantage of such an approach is that the user needs to map the high-level associations between patterns into class associations and rework his own design refinement process. In this case, the framework user has gained two things. First, he does not have to make any choices about the patterns to use because the framework developer has already made these choices on his behalf. Second, since the design is developed using patterns, the abstractions used in the pattern itself are still reflected as abstractions in the framework design; hence the framework design is still robust and maintainable.

Class-Level Instantiation

Alternatively, framework users can instantiate the refined class diagram of the framework. We showed in our paper An Object-Oriented Framework for Feedback Control Applications (1998b) how the feedback design framework (Chapter 11) is instantiated in a feedback application in the quality control line of a beverage bottle system. We point out some of the results experienced in the instantiation:

- Instantiation of a framework starting from the class diagram is difficult. This is because the framework will have several small-grained elements (classes), which makes it more difficult to understand. The user will have to understand the framework hot spots, such as where to add the implementation-specific code.

- Compared to Pattern-Level instantiation, the framework user has fewer decisions to make. This is because the framework user will not do any design refinement activities and will not change any relationships between the framework elements. He will focus only on identifying hot spots and adding implementation details.
Benefits and Limitations

Perhaps you are wondering, is POAD useful? Do I really need to build applications as composition of patterns? Can I just use any traditional OO technique? Let us take a closer look at the benefits of building applications using POAD and discuss some of the limitations as well.

Benefits

- **OO designs that result from applying the POAD development process are less subject to refactoring activities.** This is because part of the refactoring process has already been taken care of in the patterns themselves. *"Design patterns capture many of the structures that result from refactoring"* [Gamma et al. 1995, pp. 354]. In many traditional OO development methodologies you will be reworking the design and refactoring it intensively; POAD, however, uses designs that are already refactored from many successful applications.

- **Better documentation of the application/framework design.** Pattern-Level diagrams provide a high-level abstract view of the design. These abstract views act as design documentation for the application, and hence we can improve understandability of the design [Odenthal & Quibeldey-Cirkel 1997]. In many other OO development processes, packages and subsystems are used to group classes together. what goes into a package is usually an important design decision. In POAD the internals of the pattern are already known—it is its well-designed class model.

- **The risks of misusing OO design frameworks are reduced by virtue of the improved understandability of designs.** As a result of the newly introduced abstract layers and their traceability to lower design levels, we find that the design is easier to comprehend.

- **Using patterns as constructional building blocks gives an architectural view of the application because we are not concerned with implementation issues at the high-level abstract view of interfacing patterns.** However, these architecture components (fragments) are in fact design elements in terms of collaborating classes, and hence we are able to trace architecture views to lower-level design and implementation views.

- **Pattern-oriented designs are of higher quality than OO designs because they utilize design patterns that have been documented as high-quality, proven solutions to design problems.** Soukup pointed out in "Implementing Patterns" (1995) that when implementing design patterns, programmers create, extend, and modify classes throughout the software. This could create a major maintenance and traceability problem as programmers "tend to lose sight of the original patterns." The POAD approach addresses this problem by providing additional pattern-level views that are layers above the high-level design phases in which patterns are treated as design entities.

- **The POAD approach is component-based.** It shifts some of the development effort from producing designs from scratch to selecting, adapting, and assembling design fragments. On the other hand, it is not a plug, assemble, and run process, as commonly understood from the term component-based software development because the adapted and glued components are design fragments. The designer still has to work on internal details and implementation specifics.

- **POAD provides a solution to the traceability problem discussed in Chapter 2.** The solution is simply to use design models that capture the high-level view of the system, use models to capture the details, and provide a link between the high-level view and the lower-level details. In POAD the constituents of the patterns are persistent; that is, all classes are still preserved at the lower design levels.
Limitations

Despite the many benefits that POAD has, it has several drawbacks:

- Understanding patterns is a mentally exhaustive activity. To make a correct decision in choosing a pattern in the pattern-oriented design, the designer has to understand the tradeoff, motivations, forces, and related patterns. This difficulty pays off in the benefits that a designer achieves when using a selected pattern in terms of quality of the proven solution.

- To support the process of POAD, it is not sufficient to have a documentation of pattern repositories. It is essential that these pattern repositories be electronically stored in database libraries. These libraries have the traditional component library issues: component retrieval techniques and library structure.

- POAD assumes predefined pattern interfaces. The concept of pattern interfaces is new, and many people will find it strange, since they are used to using patterns at the reengineering or the class diagram level. In fact, we found that the documentation of almost all the patterns implicitly mentions something about a client object or external actor using the pattern. Examples of such interfaces are illustrated in Appendix A. We still must admit that it requires an effort to identify pattern interfaces for the existing and forthcoming literature on patterns. The principle of multiple interfaces has to be incorporated as well.
Summary

In this chapter we outline POAD from the process point of view. The following chapters elaborate on several steps in this process, and Part IV illustrates examples of applying the process. If you feel you already have enough information, you can jump to Part IV and study one of the two simple examples in Chapters 11 or 12. We also encourage you to read about the detailed process in the following chapters.
Chapter 8. Analysis Phase

Overview
Requirements Analysis
Acquaintance
Retrieval
Pattern Selection
Summary
Overview

Similar to any software development methodology, POAD starts by analyzing the application requirements. Generally, there is a strong dependency between the techniques used in the analysis process and the type of the development methodology. This is normal because the analysis process produces artifacts that will be used to design and architect the application at subsequent development phases. Therefore, the analysis process tends to produce artifacts that are most suitable for the design phase and for the rest of the development process. For instance, in traditional OO methodologies a set of analysis objects is often one of the outcomes of the analysis process. In the POAD methodology, the outcome of the analysis phase is a set of selected patterns that will be used in the application design.

In this chapter we focus on the analysis phase of the POAD methodology. In this phase we analyze the application requirements to determine the application problems to be solved and identify the patterns to be used. The inputs to this phase are the requirements as gathered from the system users or the domain experts. Another input to the analysis phase is the database of patterns, which could be a database of general-purpose design patterns or domain-specific patterns as we discuss later in the chapter.

The analysis phase contains a set of activities. The main activities within this phase are

- Requirements analysis to identify the problems to be solved and the possible breakdown of the application as a set of logical components.
- Acquaintance with relevant pattern databases to become familiar with existing solutions.
- Retrieval of patterns from the domain-specific databases to select a set of candidate patterns in an automated fashion.
- Selection of patterns from a set of candidate patterns for possible inclusion in the design process.

The purpose of this phase is to identify a set of design patterns that will be used in the application design. Starting from the application requirements and a database of design patterns, the deliverables of this phase include

- The set of patterns selected by the analyst for use in the application development.
- The rationale behind the selection of these patterns.
- The application-specific problems that the analyst identified by analyzing the application requirements.
- Documentation of why the selected patterns are anticipated to address these problems.

The analyst should not be concerned at this level with the internal details of the patterns. For example, the analyst should avoid the intellectual temptations to drill down into the pattern’s details and understand its internal class diagram, the interaction scenario between its participants, or possible variations in the solution structure. It is not required at this level to understand how the pattern will solve the problems identified from the application requirements; instead, the analyst is concerned with determining whether or not the pattern can be used and why it is better to use a specific design pattern as compared with other candidates.

Figure 8-1 illustrates the overall analysis approach.

Figure 8-1. The POAD analysis phase.
We follow the same purpose, process, and product explanations that we used in Chapter 7 to describe each activity within this phase. In addition, we added some analysis tips and guidelines for the analyst. The legend used in Figure 8-1 is the same as the one used in Chapter 7. We use a thicker border for the diagram elements that represent input and output artifacts. In the subsequent sections we elaborate on each of the four activities—requirements analysis, acquaintance, retrieval, and pattern selection—and on the artifacts produced/consumed by each. The product of this phase is a set of design patterns chosen by the application analyst.

Our discussion at this point may appear to be abstract due to the few running examples used in the description of the process or the lack of the details about the techniques used in each activity in the POAD process. We decided to use this part to describe the process without bias toward any particular application domain. Part IV includes several case studies and examples of applying the process.
Requirements Analysis

Purpose

The purpose of this activity (step) is to analyze the application requirements and identify the problems to be solved. We also determine the logical components and objects used to address these problems (components here can be thought of as packages of objects). These components are used later to facilitate the process of assessing which patterns are suitable for the design of the application at hand.

Process

Finding Components

Using the application requirements as an input, we start by analyzing the requirements to identify the problems to be solved. We look for the functionalities that the application should provide, and then we articulate the problems to be addressed. At this step, it is useful to consider use cases and develop a use case diagram in which the context of the system is established. The use case diagram shows the main use cases that capture the required functionalities and their relationships and interactions with the external actors. The use cases are documented using textual documents and realized using interaction diagrams in which logical components and objects are identified. Typically, a use case is realized by an interaction diagram showing the main flow of events and variations of this diagram to specify alternate and exceptional flows of events. These diagrams also show the internal objects and internal events needed to realize the use case. Consult the UML documentation in the literature specified in Chapter 2 for more information on use cases.

Consider the following sample from the requirements of developing a framework for a feedback control system:

*The feedback control system is required to regulate the quality control in a production line. The plant will be monitored by the system. Changes in the plant will be fed back to the control system.*

In these requirements, we identify one key use case, which is the need to monitor or observe the changes in the plant. Hence, the problem we recognize is "how to observe a plant." Plant environment sensors are specified as an external actor in the use case diagram. This use case is documented by a flow of events, which starts by a measurement object from a sensor received by the system and ends by an event where the measured object is stored in the database according to the normal flow of events. A UML sequence diagram showing the value of the measured object checked to be within the expected range of values and then stored in the database can realize the main flow of events. An alternate or exceptional flow of events can be realized in a sequence diagram specifying the flow of events when the object value is found to be out of range. Similarly, the analyst should study each requirement and determine the functionalities to be supported and the problems to be addressed.

Having identified a set of problems that we want to solve, we define logical components that will be used to solve these problems and satisfy the application requirements. We also call these components conceptual packages or conceptual components. The terms *conceptual* and *logical* are used because these are not actual application components yet; they are components from the problem space as opposed to being software components, which are often recognized as solution space artifacts.
Assigning Responsibilities to Components

We initially come up with a conceptual component definition for each use case or functionality the system will support. The analyst could also provide support for several functionalities in one conceptual component. In this case the component will play multiple roles to support more than one use case. Although a conceptual component can be used to satisfy more than one functional property of the application, it is usually simpler and easier to define fine-grained problems and functionalities and leave the clustering and grouping to later phases. For example, consider the previous requirements statement about the feedback control system. The analyst is able to recognize several use cases such as regulating or controlling the plant, “feeding back” the changes in measurements from the plant, and so on. It is preferable to treat each of these as separate use cases or functionality, and a separate problem to solve. This facilitates the process of selecting patterns in subsequent activities. As a result of this activity, we identify components from the problem space for which we assign functional responsibilities.

Relationship to Software Architecture

Consider the similarity between this activity and the process of defining the application architecture in any software development methodology. Software architecture description (definition) techniques include several activities, such as

- Identifying a set of application components and their functional responsibilities.
- Defining the interconnection and relationships between the application components.
- Analyzing the behavior of the system as collaboration of interacting components.

In the analysis phase of POAD we are more concerned with the first task: identifying components and their responsibilities. These functional responsibilities define the overall roles of a conceptual component. They are used to guide the process of pattern selection by identifying the necessary responsibilities required from a particular pattern that will be used to implement the conceptual component. However, there is one difference between components in application architecture and conceptual components in POAD. Whereas the earlier is defined in the application solution space, the latter is defined in the problem space. Software architecture components are actual, realizable, deliverable (and sometimes executable) units. POAD components are more abstract in the sense that we use them to select and identify the patterns to use in the application design, while they themselves are not used in the design.

Analyzing Large Applications

This activity is more challenging for large software systems. For those systems, it is not possible to consider all the problems and functionalities in the analysis activity at the same time. It is usually difficult to identify all components at the same level and at the same time. Every software development methodology provides support for grouping and clustering techniques. Analysts usually consider grouping mechanisms to break down the problem domain into manageable pieces and to provide hierarchical views of the system. Some refer to these grouping constructs as subsystems; others refer to them as packages and frameworks. In general, it is a recommended practice to break down the complex system into grouping units; let's call them subsystems. For each subsystem, an analysis procedure is conducted to identify individual components within the boundary of that subsystem and their responsibilities. In POAD we take a similar approach: large systems are broken down into manageable subsystems and each individual subsystem is analyzed to determine the design components and the functionalities provided by that subsystem. Chapters 13 and 14 illustrate the application of the POAD process to such systems.

Iterating with Other Analysis Activities
During the analysis process in POAD, a blind definition of conceptual components will not work in practice. This is because it is difficult to find a pattern (or a set of patterns) that implements the predefined functionalities of these components. This top-down approach to break the system into subsystems and then each subsystem into a set of conceptual components is suitable for a first iteration in order to define the problems that we need to solve and the functionalities to support.

Analysts and designers tend to look for ideas to borrow from systems they analyzed, designed, and developed earlier. If they are unable to find similar applications, they turn to someone who solved similar problems before—for example, domain experts. If they fail to find domain experts, they look for documented solutions in libraries or databases. They get acquainted with the solutions that solved problems similar to the one they are considering. This is called the acquaintance activity, as discussed in the next section. The POAD requirements-analysis activity usually relies on the acquaintance activity, and there are usually several iterations between the two activities.

**Product**

The product of this step is a definition of a set of conceptual components (packages), their use cases, and their responsibilities.

**Analysis Tips**

When performing requirements analysis in POAD, consider the following guidelines:

- **Do not get stuck with requirements analysis.** The activities within the analysis phase are generally iterative. The analyst should not get stuck at one particular activity looking for perfection in defining components. The analyst should keep in mind that iteration is essential between defining the problems, the components, and their responsibilities and other activities within the analysis phase, specifically the acquaintance activity. If it is hard to proceed with identifying more components, it is better to move to the next activity and revisit the requirements-analysis activity later.

- **Group patterns into frameworks or subsystems.** For large, complex systems, the analyst should consider managing the complexity of the system by grouping components using some grouping mechanism like subsystems or packages. This will facilitate the analysis of each subsystem rather than working on the overall application as one flat architecture.
Acquaintance

Purpose

The purpose of the acquaintance activity is to get the analyst familiar with existing solutions in the application domain. The objective is to help the analyst identify a realizable set of components that satisfy the application requirements, that is, a set of components that can be implemented using existing real and concrete solutions.

Process

In POAD the main objective of the analysis phase is to identify a set of patterns to be used in designing the application. This assumes the existence of pattern libraries from which the analyst can select patterns. These libraries could be domain-specific pattern libraries or general-purpose pattern libraries. As an example of domain-specific pattern libraries, consider a pattern library for the telecommunication systems. Such a library would include patterns that are specific for the telecommunication domain, such as the Adaptive Communication Environment (ACE) patterns [Schmidt et al., 2000], which include the Acceptor, Connector, Service Handler, and other patterns. As an example of a general-purpose pattern library, consider the classic patterns by the GoF [Gamma et al., 1995], which are useful in various OO applications. Such a library would include patterns such as the Observer, Strategy, and Factory patterns. Such classification of pattern libraries would require a definition of what constitutes a domain and the scope of applications within that domain [Mili et al., 1999].

Pattern Database Activities

During the requirements-analysis activity, the analyst defines a set of components that satisfy the application requirements. In POAD, those components are implemented using design patterns, which are retrieved from either a domain-specific or general-purpose database of patterns. Therefore, during the process of defining these components, the analyst should consider which patterns already exist in the pattern database. This will help in identifying components that are realizable by existing design patterns and hence make the POAD process a viable approach.

The POAD process is a reuse-based process. We reuse existing design solutions in the form of design patterns. We cannot simply focus on the problems that we identify from the application requirements and ignore existing solutions. It is essential for the analyst to use existing solution fragments in building the overall solution. In POAD those solution fragments are patterns that are stored in pattern databases. Similar to traditional software component library management, there are two activities associated with the pattern database management, and they are important to the POAD approach: the acquaintance and retrieval activities.

What Is Acquaintance?

The acquaintance activity is the process by which the analyst gets himself acquainted with existing patterns in the database. During this
process, the analyst gains some knowledge about existing solutions in the application domain. This is often the case even in traditional software development approaches. As an example, consider a C++ developer who is developing an application for image processing. Instead of redeveloping his application from scratch, the developer usually considers existing solutions first. Most C++ IDEs are equipped with a library of C++ classes that the developer can use—for example, the basic data structures (queues, lists) and math libraries. The application developer often visits this general-purpose library for C++ classes that are ready to use and uses these existing solutions to drive the overall application solution. As for domain-specific libraries, the developer would also consider browsing existing off-the-shelf image processing libraries for solution fragments to use.

A similar approach is considered by POAD for off-the-shelf design pattern libraries. The analyst browses existing pattern databases for solution fragments (patterns) to use in developing the overall application design. There is a difference, though, between the example we mentioned above (reusing C++ components) and POAD (reusing design patterns). Components that are distributed with an IDE environment are usually code or executable components that can be used directly in developing the application code. Design patterns in POAD are used as design components that require further refinement and composition to produce the OO design of the application prior to generating or developing code.

What versus How!

The acquaintance activity includes browsing catalogs of patterns that are stored in the library for the purpose of understanding existing patterns and their usage. It is necessary at this phase to consider which patterns solve the current problems. The analyst should avoid being trapped in the process of finding out how the pattern solves the problem. At this level, what matters is what can be reused, not how to reuse it.

The pattern documentation is usually very rich in information. It has many sections related to the intent of the pattern and what problem it solves. The pattern documentation also provides details about the structure of the pattern solution and the dynamics between the pattern constituents. The documentation takes several forms, such as the GoF or the Alexandrian templates. The analyst should focus at this level on the sections of the pattern that will help in understanding what this design solution is about and not the details of how the solution will be used. For instance, the analyst should focus on the problem and intent section and not on the participants, dynamics, or structure sections.

Rationale and Example

The rationale behind the acquaintance activity is to attract the analyst toward existing good solutions. Some of the patterns in the library may motivate the analyst to identify conceptual components that are easily designed and implemented using existing patterns. For instance, consider the case in which the analyst is analyzing a reactive system or a real-time system. Due to the reactive nature, those applications are heavily state-based. When the analyst is performing the acquaintance activity, the existence of a pattern language of state machines [Dyson & Anderson 1998; Yacoub & Ammar 1999] and statecharts [Yacoub & Ammar 1998b] would motivate the analyst to develop a model (or an architecture) that is based on collaborating components whose behaviors are substantially modeled in statechart specifications. For example, the ROOM design approach encourages the development of architecture models that are in the form of collaborating state machines [ROOM 1994]. Those systems can be easily implemented using the state machine and statechart patterns, and these patterns will be the design components to use in building the application design.

Iteration

The acquaintance activity and the requirements-analysis activity are heavily tied to each other. It is expected in POAD that there will be several iterations between the acquaintance activity and the requirements-analysis activity. This is because during the acquaintance activity the analyst gains the knowledge of existing solutions in the solution space, and during the requirements-analysis activity the analyst gains the knowledge of the existing problems and the required functionality in the problem space. Hence, to define components that satisfy requirements and at the same time are realizable using patterns, the analyst has to define components that solve the domain problems and are realizable by off-the-shelf patterns. Therefore, iteration is encouraged between the two activities, as shown in Figure.
Product

The outcome from this activity is not a concrete artifact. The outcome is the knowledge gained by the analyst about existing solutions that would help him in the requirements-analysis activity. This knowledge will help the analyst in defining components that can be realized using patterns from the pattern databases.

Analysis Tips

When performing the acquaintance activity in POAD, consider the following guidelines:

- **Don't study the whole library.** It is not the objective of this activity to analyze and study all the patterns in the library that are applicable in the application domain. In the analysis phase, the analyst is mainly interested in analyzing the problems and defining the components that can provide solutions to these problems. The acquaintance activity helps the analyst define realistic components instead of components that would be hard to implement. Therefore, a quick grasp of the pattern library is useful, but spending the time to study each and every pattern in the library is not desirable.

- **Iterate.** We keep emphasizing the role of iteration in every step in the analysis phase because it is an important key to the success of the application of a reuse-based development approach in general and to the POAD approach in particular. Iteration between the requirements-analysis activity and the acquaintance activity is a useful analysis phenomenon. The acquaintance activity mainly helps the requirements-analysis activity. The analyst is expected to iterate several times between the solution domain and the problem domain.

- **Focus on the problem.** At the analysis level we should focus on the **intent** and **problem** sections of the pattern documentation. At this level, we should not be considering any details about the pattern solution structure or dynamics. Instead, the analyst should get herself acquainted with the high-level description of what the pattern does, since this is always an easy way to find out whether these patterns are applicable in the application domain under consideration.
Retrieval

Purpose

The purpose of this activity is to retrieve a set of relevant patterns from the pattern databases. The retrieved patterns constitute a set of candidate patterns that the analyst might consider during the selection activity.

Process

The retrieval activity focuses on defining how to select a pattern from a catalog of design patterns. The catalog is usually stored in the form of a pattern database, whether a general-purpose or domain-specific database. As mentioned earlier, these databases are assumed as an input to the analysis phase. There are two activities related to pattern databases: acquaintance and retrieval. Acquaintance is defined as browsing the patterns stored in the database. Retrieval is defined as selecting patterns from the database. The process of retrieving patterns from a pattern database is similar to the problem of retrieving software assets from asset libraries, where the asset is the design pattern and the library is the pattern databases.

Software Asset Libraries

The topic of software assets retrieval from libraries has been an important research field in computer science for decades [Mili et al., 2001]. The huge effort expended on the topic has resulted in a diversity of techniques and methods for software asset representation and retrieval. A comprehensive survey of existing retrieval techniques is discussed by Mili and colleagues in "A Survey of Software Reuse Libraries" (1998). Most of these retrieval techniques have been proposed for software assets in general, whether specification, design cliché, or code assets. Unfortunately, up to the time of writing this book we have found no thorough study of the application of research results from the field of software asset retrieval to design patterns' representation, storage, and retrieval.

The templates used to document a design pattern are very information-rich. They introduce new dimensions that should be considered in the traditional software asset representation and asset retrieval from libraries. The few templates that are used to document a design pattern have proven an effective means of communicating and reusing those design solutions between application designers and developers. Those pattern documentation templates such as the GoF, POSA, and Alexandrian templates, should be considered a new means of representing software assets in asset libraries in general and for pattern representation in databases in particular.

On the other hand, the effort to build a pattern library should make use of the results and experiences gained by the designers of software asset libraries over the last several decades. Researchers and practitioners have identified several issues when building software asset libraries. In the following paragraphs, we give a brief description of some of these issues to give the analyst examples of what activities should be considered for pattern retrieval.

The Database Query

Retrieval depends on matching a set of candidate assets against a user-defined query. The representation of the query and the asset is an important consideration. There are several ways to represent queries, including information retrieval methods such as natural
language methods, descriptive retrieval methods such as a list of keywords, denotation semantics methods such as the input/output signature or function description, and structural methods such as the program or design structure. There is even more diverse representation of the asset itself, since it usually depends on how the asset is abstracted in some representation form.

There are two important activities within the retrieval process: navigation and matching. The navigation technique defines how the assets in the library are visited. This depends on how the assets in the library are organized—the storage structure. For example, the navigation could be random if the assets are stored in flat organization. The matching technique involves testing to determine whether the asset is relevant and should be selected. There are two conditions that can be used by the matching activity: relevance, which is the condition under which an asset is considered to be relevant with respect to the query, and matching, which is the condition under which the asset is selected. In practice you can use these two conditions to distinguish exact matches and relevant matches, to control the retrieval scope, and to categorize the results of the retrieval process.

Retrieval Goals

Every retrieval technique should have a retrieval goal. For example, the retrieval goal could be to retrieve all patterns that can implement an observing or monitoring strategy. It could also be to retrieve all patterns that have a subject and observer as interfaces. Having the retrieval goal defined, you can then assess the precision and recall of the retrieval technique. Precision means counting the number of relevant assets in the query result as opposed to the total number of assets retrieved by the query. Recall means counting the number of assets retrieved as opposed to those that are in the library and were not retrieved by the retrieval technique.

Methods and techniques to store and retrieve a software asset in a library are plentiful. To distinguish and compare those techniques, we have to define the assessment criteria, which should include features about the library itself, the asset, and the user query. Such criteria include the nature of the asset (code, executable, design, etc.), the representation of the query, the representation of the asset, the structure of the library, the navigation schema, the retrieval goal, the relevance criterion, and the matching criterion. These criteria are further discussed in Mili and colleagues' “A Survey of Software Reuse Libraries.” Further research is essential to assess how design patterns, as library components, can influence the software component library structure, retrieval methods, and matching criteria, and how the previous work on these topics could be applied in the design pattern field.

A True Pattern Library

So far, we cannot say that a true pattern library does exist. We can easily find some collection of patterns (such as the GoF patterns) distributed as a standalone package or as a module distributed with a development IDE environment or a modeling environment (such as Rational Rose). However, we cannot find a library that has the huge collection of patterns that were produced over the years in PLoP conferences and PLoPD books. Such a true pattern library should be kept in a persistent location or database and should provide easy access to the content—for example, through an online web-based interface. Prior to the construction of such a true pattern library, we have to study possible classification criteria for pattern types and their domains and applications. Moreover, this true library should be updated as new patterns are shepherded in the yearly PLoP conferences. The indexing of existing design patterns as developed by Linda Rising [Rising 2000] is a significant start toward achieving the ultimate goal of a pattern database. Until such a database becomes a reality, the analyst will have to face the reality that he will be using propriety databases and pattern collections as inputs to the retrieval activity in the POAD approach.

Practical Guidelines for Matching Criteria

Following are some guidelines to help the analyst select patterns from pattern databases or pattern collections. These guidelines are helpful in the development of a matching criterion that is used in constructing a query to the pattern database.

- **Problem matching.** At the analysis phase, we look for patterns that solve a particular problem. Hence, we should focus on searching the pattern catalog using matching criteria that are based on matching the intent, problem, or context sections of the pattern.
Applicability. Many pattern documentation formats have an applicability or forces section that describes the tension between solutions and how the pattern solves that tension. Checking this applicability/forces section could give an early assessment of whether the pattern is suitable for the application under consideration.

Consequences. An important section in pattern documentation is the consequences of using this pattern. Sometimes the analyst may find the resulting context (or consequences of applying the pattern) unsuitable for the application. For instance, assume the analyst is analyzing a distributed system and the resulting context of applying a communication pattern is a penalty in the performance. In this case, the analyst might want to consider alternative design patterns because performance is an essential quality attribute in the application.

Related patterns. The analyst should check the related pattern section in the pattern documentation as a source for other relevant patterns, which may provide similar or better solutions. The pattern documentation has a section about other patterns that are related to the current pattern. This section might open opportunities to check other patterns that solve problems related to the problem at hand.

Conflicting patterns. In many cases the use of a pattern may contradict with the use of another pattern. This is usually mentioned in the pattern documentation template under the related pattern section. Hence, the analyst should consider checking for the use of contradicting patterns, which will help in excluding candidate patterns at an early stage.

Product

The outcome of this process is a set of candidate design patterns. Those candidate design patterns are used as input to the selection activity to select the patterns to pass to the design phase.

Analysis Tips

When performing the retrieval activity in POAD, consider the following guidelines:

Tend to include and not to exclude. During the retrieval activity, the size of the set of candidate patterns is heavily dependent on the matching and retrieval criteria. The analyst is then faced with the problem of selecting pattern candidates. In this process, if the analyst is hesitant about selecting a pattern, he should add the pattern to the candidate list rather than excluding it. During the pattern selection activity or during the design phase, those patterns will be reassessed based on more detailed knowledge of the problem and the solution.

Check the literature. The analyst should keep updating the design pattern collection used in her application domain by checking the yearly documentation of design patterns. The availability of a good collection of design patterns is a key factor in the success of the POAD approach.
Pattern Selection

Purpose

The purpose of this activity is to select a set of patterns that fulfill the responsibilities of each conceptual component that was identified from the application requirements. Those selected patterns are then delivered to the design phase of the POAD process to be used in building the pattern-level design views.

Process

Problem-Solution Matching

The pattern selection process is more like a problem-solution matching technique. The problems are defined by the set of components and their responsibilities as defined from the requirements-analysis activity. The solutions are defined by the set of candidate patterns as provided by the retrieval activity. The selection process is then the process of matching solutions (design patterns) to the problems and responsibilities of the conceptual components.

The GoF proposed several tips on how to select a pattern from their catalog of 23 general-purpose patterns [Gamma et al. 1995]. The proposed tips rely on designer intuition and reasoning:

- Consider how the pattern solves the design problem by scanning the solution section and assessing whether the solution provided can be used in solving the problem at hand.
- Scan the intent section, which often describes what the problem is and hence could be matched to the problem at hand.
- Study how patterns interrelate. Look for similar patterns or patterns that are normally used in pairs or groups.

Selection Is Difficult

Selecting a pattern to match component responsibilities is a difficult task even when the selection process is concerned with a small-scale library of patterns. The difficulty in selecting a pattern lies in several dimensions, which include:

- The selection process is human-intensive. A design pattern is a building unit that provides a solution to a common problem that has been identified from the experiences in developing several other applications. The problem that the pattern solves is often captured from the mind of the designer and articulated in a pattern format. A considerable amount of human effort,
reasoning, and intuition is required to understand the problem and the solution provided in the pattern documentation and to match them to the problem and solution required in the design of the new application.

- **Maturity**: Development methodologies that use design patterns as reusable OO designs are not in as solid and mature a phase as other software development methodologies that have been around for years. For example, few libraries of patterns exist, and the use of design patterns is still not a common practice among software designers and developers.

We are not aware of any analytical and systematic research for automating the process of retrieval and storage of design patterns from and in databases. Some commercial and shareware tools (as discussed in Chapter 16) support search and retrieval of patterns, but the process is usually based on a simple keyword matching techniques. Therefore, the final selection process is usually left to the analyst.

### Iteration

During the selection process, the analyst revisits the acquaintance and retrieval activities for refinement and iteration. The outcome of the retrieval activity is a set of candidate patterns. If none of the candidate patterns satisfy the responsibilities of the required components, the analyst has to add more patterns in the candidacy pool. This is possible by getting the analyst more acquainted with the pattern database and its content. It is also possible by relaxing the constraints on the pattern-matching criterion or the relevance criterion in the retrieval technique. Therefore, there will be several iteration loops between the selection activity and the retrieval and acquaintance activities before the final selection of patterns is made.

### Using Domain-Specific Libraries

To make the selection process easier and more effective, a domain-specific library of patterns should be used instead of a general-purpose one. This will ease the acquaintance process, since the analyst will be able to analyze fewer patterns that are more relevant. For example, when developing a distributed application, it is helpful to look for patterns in the field of distributed systems and telecommunications rather than in a general-purpose database that would include patterns from multiple fields. Similarly, when developing reactive systems, we should consider patterns in this field, such as State machine patterns and Event Handler patterns. Unfortunately, the classification and dissection of patterns and their relevance to application types is not currently a reality. As more people start using systematic development methodologies (such as POAD) to develop software using patterns, the classification issue will become more important, and a concrete solution will be proposed.

### Study Relationships Between Patterns

The increasing numbers of patterns and pattern languages have prompted the development of classification techniques for patterns and pattern relationships. This goes back to the first book on patterns (the GoF book) in which classification of creational, structural, and behavioral patterns was introduced. A pattern map was also developed showing the relationship between the 23 patterns described in the book. Since the text of each pattern should describe its relationship to other patterns, the pattern literature shows different classification schema of these relationships, which can be inconsistent. James Noble developed a classification scheme for the relationships between patterns [Noble 1998]. Three primary relationships and a number of secondary relationships between patterns are identified. The primary relationships are as follows:

- a pattern uses another pattern.
- a pattern refines another pattern.
- a pattern conflicts with another pattern.

Typically, more complex patterns use simpler patterns. For example, the Model-View-Controller (MVC) pattern [Buschmann et al. 1996]
uses instances of the Observer pattern, the Strategy pattern, and the Composite pattern. The "refined" relationship can be used to define a new pattern as a refinement of a published pattern. A conflicting relationship shows that instances of these patterns should not be used together; that is, they provide mutually exclusive solutions to the same problem, and it is not possible to use them interchangeably. Noble also discusses a number of secondary relationships and shows how they could be defined using the primary relationships. This classification scheme helps in the process of selecting patterns from pattern databases.

Product

The outcome of this activity is the set of patterns that will be used in designing the application.

Analysis Tips

When performing the pattern selection activity in POAD, consider the following guidelines:

- **Don't try to solve the problem.** In many cases the analyst is tempted to drill down into the details of the solution of the candidate patterns and find out how they solve the problem. This could happen by either dwelling on the details of the pattern or by trying to instantiate the internal design of the pattern and apply it to the specific application under consideration. This is not a good practice, since it consumes the analyst's time and might cause her to miss the overall picture—as well as the project deadline. Whereas the analyst's responsibility is to determine whether the pattern can be used, it is the designer's responsibility to instantiate the pattern and to apply it in designing the application.

- **Document the selection decisions.** The analyst should document every selection decision made during this activity. For instance, it is useful to include with the selected patterns the rationale behind the selection, what problem in the application this pattern will address, and why the pattern is anticipated to solve some application design problems.

- **Iterate.** As always, iteration is very important to the success of the analysis phase. The analyst should iterate between the selection activity and each of the acquaintance and retrieval activities.
Summary

In this chapter, we discussed the analysis phase of the POAD methodology. The analysis phase is a reuse-based analysis process that is a combination of two approaches:

- **Top-down analysis** in which we identify the problem from the requirements space and search for design patterns that provide solutions to that problem.
- **Bottom-up analysis** in which we know of some useful solutions in the domain under investigation and we investigate whether these solutions can be used to satisfy the user requirements.

The outcome of the analysis phase is the set of patterns that are selected by the analyst to be used in designing the application. Each of the selected patterns is tied to specific problems in the application domain. Documentation is an essential artifact that should be produced during the analysis phase. The documentation developed from the analysis process should cover:

- The design components that are derived from the application requirements.
- The functionalities and responsibilities assigned to these components.
- The selected patterns that could be used to implement the component.
- The problem solved by each of the selected patterns and the rationale behind the selection of those patterns to solve that particular problem.

The documentation and the selected patterns that are produced as a result of the analysis phase will then be used at the design phase to develop the pattern design diagram for the application, as explained in the following chapters.

The analysis process that we illustrated in this chapter provides an overview of the analyst's tasks without defining the exact techniques to use in each activity. For instance, we discussed the retrieval activity, but not which retrieval criteria is suitable for pattern databases. Similarly, we discussed the identification of conceptual components and their responsibilities without defining techniques that the analyst should use for that purpose, such as use cases or analysis scenarios. There are two reasons for such abstraction and possibly ambiguity. The first reason is to give the analyst the flexibility to conduct the activity using techniques that best suit his analysis skills. The POAD process is not concerned with the details of the techniques and algorithms used by each analyst as long as they fall within the analysis framework and serve the purposes of each activity. The second reason is that the comparison of the best techniques to use under each activity is a subject of a research on its own, and more experiments are needed to assess the suitability of such techniques. This chapter reviewed the overall picture and the activities required in the process. We hope that the examples in Part IV will help you understand what it takes to conduct these analysis activities in practice.
Chapter 9. Design Phase

Overview

Constructing Pattern-Level Models

Constructing Pattern-Level with Interfaces Models

Constructing Detailed Pattern-Level Models

Summary
Overview

In this chapter we discuss the design phase of the POAD methodology. During the design phase, we create the initial design of the application using patterns as building units. The inputs to this phase are the patterns selected by the analyst during the analysis phase. In addition, all the analysis documentation produced during the analysis process will be used by the designer to glue these patterns together to develop the application design. This documentation contains information about how each pattern solves a particular design problem as well as the initial break down of the application into conceptual components.

The design phase has three main activities:

- Constructing the Pattern-Level diagram, which provides a view of the design as a high-level composition of patterns.
- Constructing the Pattern-Level with Interfaces diagram, which provides a detailed view of the relationships and interconnections between interfaces of the selected patterns.
- Constructing the Detailed Pattern-Level diagram, which provides a detailed view of the internal design of the selected patterns.

During the design phase, a set of design models is created. The type of these models as well as the elements used in these models are described in detail in Chapter 5 and are used in this chapter to develop the pattern-oriented design of the application.

The purpose of this phase is to create an initial design of the application, using patterns as the primary building blocks. Starting from a set of patterns selected by the analyst, the deliverables of the design phase include

- A high level view of the application design in the form of one or more Pattern-Level diagrams.
- The Pattern-Level with Interfaces diagrams, which capture the interface relationship between the selected patterns.
- The Detailed-Pattern Level diagrams, which provide the application level details of the internal design of the patterns.

As we discussed in Chapter 5, all the design models produced at this phase should be preserved. A traceability mechanism would allow the designer to traverse the design models up and down the design abstraction hierarchy. This traceability mechanism is a function that should be supported by a POAD development environment (see Chapter 16).

Figure 9-1 illustrates the overall design phase. The legend used in Figure 9-1 is the same as that in Chapter 7.

Figure 9-1. The POAD design phase process.
In the subsequent sections we elaborate on each of the three main activities in the design process: constructing the Pattern-Level diagram, constructing the Pattern-Level with Interfaces diagram, and finally constructing the Detailed Pattern-Level diagram.
Constructing Pattern-Level Models

Purpose

The purpose of this step is to create the pattern-level models for the application. These design models are used to express the application as a composition of constructional design patterns.

Process

The outcome of the analysis phase is a set of patterns that the analyst selected to provide the application functionalities required by the user. Those patterns are used by the designer to develop a high-level view of the application as a composition of patterns. The creation of this high-level view involves several steps that include creating pattern instances, identifying relationships between these instances, and developing the Pattern-Level diagram.

Creating Pattern Instances

In this step, the designer creates an instance for each pattern that is selected by the analyst. Instantiation of a selected design pattern means turning a design pattern into a tangible design artifact. The instantiation process involves describing the patterns and their constituents in an application-specific context suitable for modeling the application at hand. The instantiation process includes two steps. The first is identifying the name and type of the pattern and is performed during the design phase. The second is related to giving application-specific names to the pattern internals and is performed during the design refinement phase.

At this level, we are interested in the application-specific problem that each pattern solves. Therefore, we define the instance of each pattern using a name and a type. The name is related to the application-specific problem for which the pattern is used. Therefore, the name distinguishes the pattern in that particular application design. The type of the pattern is the well-known pattern name that is commonly used to qualify the pattern in any pattern description (documentation) template. The type is the universal name for the pattern used by many application designers.

For illustration, we consider a concrete example drawn from using POAD to develop the application design of a feedback control system. In this application, we assume that we want to observe the changes in the readings from a specific sensor in the plant to be controlled by the application. A software component is used to interface to the sensor and to represent the readings obtained on a periodic basis. For this application, the analyst has chosen to use an Observer pattern to perform the function of observing this sensor. In order to instantiate the Observer pattern in this application design, we define two things for the selected pattern: its application-specific name and its type (more application-specific details will be required at later phases). The type of the pattern is Observer, since this is the common pattern name used to qualify the Observer pattern [Gamma et al. 1995]. As for the application-specific name of the Observer pattern instance, the designer will give it a name related to the application-specific problem it is used to solve. For the feedback control system, the designer may want to consider the name SensorObserver, for instance. In this case, we say that the application design will contain a pattern instance called SensorObserver of type Observer.

Multiple Instances of the Same Type
The designer may want to use multiple instances of the same pattern type in developing the design of an application. The pattern instance name is used to distinguish which pattern the designer refers to in the design models. For example, assume that for the same feedback application discussed earlier, the analyst decides to use another Observer pattern to observe the changes in the input values from the user. The user input in a feedback control system is usually compared with some feedback measurements, and the result is used to control the plant. In this case, the application design will contain another pattern instance called InputObserver of type Observer pattern. The feedback application design now contains two pattern instances: InputObserver and SensorObserver of the same Observer type. Therefore, it is normal that the design may contain several instances of the same pattern type; each is used to satisfy a different application requirement.

Defining Pattern Instance Relationships

Having identified the pattern instances, the next step is to define how these instances are related to each other—that is, the relationships between pattern instances. In POAD the relationships between patterns are defined as dependencies. The semantic of a dependency relationship used between patterns has a uses meaning. In Chapters 5, 6, and 15 we elaborate on the different types of dependencies and how pattern dependencies are related to UML semantics and the metamodel.

The designer then determines whether or not there is a relationship between two specific pattern instances. He also studies the direction of each dependency relationship. For example, consider the feedback control system in which the designer has selected to use an InputObserver instance of type Observer pattern to observe the user input to the system. He also selected to use a ForwardStrategy instance of type Strategy pattern to provide a flexible mechanism to hook various strategies for controlling the plant. When defining the relationships between pattern instances, we find that the output from the InputObserver will be used to trigger a control strategy to be applied to the plant. Hence, the InputObserver instance will use (will forward its output to or invoke) the ForwardStrategy instance, so the designer creates a dependency relationship from the InputObserver to the ForwardStrategy instances.

It could be difficult to understand the relationship between pattern instances by just studying the static aspects of the application. Dynamic analysis of the application could be helpful in identifying the relationships between pattern instances. For example, the designer might consider using a UML scenario or interaction diagrams to capture the sequence of invocation of design components, and hence the dependency relationship between pattern instances could be developed. Other designers could use the documentation produced from the analysis phase, which includes the description of the conceptual components and their functionalities. There are several other techniques that the designer can use to study the dynamics of an application, such as studying the behavior using state machines or object diagrams.

Develop the Pattern-Level Diagrams

During the process of creating pattern instances and defining their relationships, the designer might start developing the Pattern-Level diagrams, which capture pattern instances and their relationships, as discussed in Chapter 5. We use UML models to represent Pattern-Level diagrams where stereotyped packages are used to syntactically (for semantic support, see Chapter 15) represent patterns, and UML dependency relationships are used to represent pattern instance relationships. For example, consider the feedback control application, which has two pattern instances: InputObserver of type Observer and ForwardStrategy of type Strategy. The InputObserver instance uses the ForwardStrategy instance, as discussed earlier. Figure 9-2 illustrates part of the Pattern-Level diagram for this example. We note that this is an incomplete design diagram for the example; the complete Pattern-Level diagram of the feedback control application framework is developed in Chapter 11.

Figure 9-2. A simple example of a Pattern-Level diagram for the feedback control system.
Subsystems and Frameworks

The designer may construct one Pattern-Level diagram for the overall application. For large complex applications, it is wise to divide the application into a set of subsystems or frameworks. In this case the designer constructs a Pattern-Level diagram for each individual framework. Incorporating the concepts of design framework and subsystems is useful for cases in which parts of the system cannot be analyzed in terms of patterns due to the limitations in the available catalogs of patterns. In such cases, these frameworks are incorporated in the Pattern-Level design view as UML packages and are further decomposed at lower design levels in traditional OO models such as class diagrams.

Product

The product of this process is the pattern-level model for the application and for each individual subsystem or framework.

Design Tips

When constructing the Pattern-Level diagram in POAD, consider the following guidelines:

- **A pattern list is not enough.** The designer will not be able to develop the Pattern-Level diagram using the set (or list) of selected patterns because the list by itself is not sufficient. It is the documentation that is produced from the analysis phase that adds the application-specific meaning to a pattern. The designer should use the analysis results, including the definition of design components and how the selected patterns are anticipated to solve the design problems. When the designer of the application is not the same as the analyst, the documentation becomes an important knowledge-transfer mechanism between the analyst and the designer.

- **Study the dynamics.** Studying the application dynamics will help the analyst identify the dependency relationships between pattern instances. However, the designer should consider the dynamics between the patterns as building blocks—not the dynamics of the internals of each block. Studying the dynamics of the internal design of the pattern introduces another level of difficulty that is not needed during this activity. The focus at this level is on the patterns and how they interact. The focus is not on the internal elements of a pattern.

- **Meaningful names.** When choosing application-specific names for the pattern instance, the designer should choose names that are meaningful in the application context. When there are multiple instances of the same pattern, the designer has to choose unique names to avoid naming conflicts. We have found that using the pattern type as part of the name makes the design more understandable. For example, if we want to use an Observer to a sensor-observing function, we choose the instance name SensorObserver.

- **Revisit the analysis phase.** In some applications, the designer might find that the patterns selected by the analyst are not suitable or cannot be used in developing the Pattern-Level diagram. In this case, the designer might want to work with the analyst to revisit the decisions made about the selection of specific patterns.
Constructing Pattern-Level with Interfaces Models

Purpose

The purpose of this activity is to analyze the relationships between pattern instances and define the relationships between their interfaces.

Process

The dependency relationship between patterns in the pattern-level view is a conceptual high-level dependency that illustrates which patterns use the others. To develop lower-level design models for the application, these dependencies should be further traced to lower-level design elements that will be further used to glue the internal participants of a pattern with the internal participants of another pattern. It is the purpose of this activity to develop Pattern-Level with Interfaces diagrams that will help the process of tracing these dependency relationships to lower-level class relationships. The Pattern-Level with Interfaces diagrams provide an intermediate step to bridge the gap between various design abstraction levels, where the higher abstraction level is the pattern instance dependency relationships and the lower abstraction level is the class association relationships between pattern participants.

To be able to analyze the dependency relationship between pattern instances, we first expose the interfaces for each pattern instance and then create relationships between their interfaces.

Exposing the Pattern Instance Interfaces

As we discussed in Chapter 4, POAD uses specific types of patterns, the constructional design patterns. Constructional design patterns are design components with interfaces. Pattern interfaces can be either interface operations or interface classes.

The first step to analyze the dependency relationship between pattern instances is to identify the interfaces of each pattern. The interfaces of a pattern are the hooks by which the pattern can be integrated in the design with other design artifacts, including other patterns. Each pattern in POAD comes with a predefined set of interfaces. At this phase, the designer reveals the predefined pattern interfaces to be used in hooking them to other patterns.

The interface definition of a design pattern is usually buried in the pattern documentation. There is no explicit field in most pattern documentation templates that defines the interfaces or hooks by which a pattern can be composed with other design artifacts. In many pattern documentations, we sometimes see a reference to a "client" class that invokes some method or has a reference to a specific participant in the pattern. Those methods and pattern participants can then be considered interfaces for the internal design of the pattern because they are the parts of the design that appear to a client. POAD makes explicit use of pattern interfaces in modeling the application design as a composition of patterns. It is expected that the pattern database used for the POAD methodology will contain documentation about the pattern interfaces. Examples of interfaces for some well-known patterns are illustrated in Appendix B.

A pattern could have multiple interfaces. Patterns are reengineered design solutions that have been used in several applications to solve common design problems. The internal design of the pattern is abstracted from existing application designs. Similarly, pattern interfaces are extracted from existing application designs by considering how the rest of the design uses the pattern. It might happen that different
application designs interface with the pattern in different ways. Therefore, we can expect a pattern to have multiple interfaces. For example, consider the Strategy pattern [Gamma et al. 1995]. One designer can consider the Context class as the interface class for the pattern. Another designer may consider the contextInterface() method to be the interface. In this case, a pattern would have multiple possible interfaces. The designer is provided with all possible interfaces during design time. It is up to the designer to select the interface that most suits the current application design.

Identifying Relationships Between Pattern Instance Interfaces

In this step, the dependency relationships between pattern instances in a Pattern-Level diagram are further translated into relationships between classes and/or operations interfaces of each pattern instance. The designer will have to make design decisions about which pattern interface to use. The designer will also construct the relationship between the pattern interfaces that will implement the high-level dependency relationship between pattern instances.

In POAD we use the UML interface syntax to represent a pattern interface, which could be an interface operation or an interface class. We also use UML dependency relationship to express the relationship between the interface operation (or interface class) of one pattern and the other. The pattern instance is represented as a stereotype package, which uses the UML "realizes" relationship to indicate that the pattern implements a specific interface.

As an example, consider the feedback control application and its Pattern-Level diagram illustrated earlier in Figure 9-2. The InputObserver pattern instance uses the ForwardStrategy pattern instance. In order to examine the detailed relationship between these two instances, we first expose the interfaces for each pattern instance. The InputObserver instance is of type Observer pattern for which we consider the two interface operations Update() and Notify(). The ForwardStrategy instance is of type Strategy pattern for which we consider the interface class Context. In this application, as soon as the input values change, the control strategy should be applied to control the plant. Thus, the Update() operation of the InputObserver instance will invoke the Context interface of the ForwardStrategy instance. Hence, the Update() interface will be connected to the Context interface using a dependency relationship. Figure 9-3 illustrates the Pattern-Level with Interface diagram for this simple example.

Figure 9-3. A simple example for the Pattern-Level with Interfaces diagram of the feedback control application.

We note that in this example the Notify() interface for the InputObserver instance does not play any role in the dependency relationship between the InputObserver instance and the ForwardStrategy instance. It might play a role in the relationship between the InputObserver instance and other pattern instances in the design.
This process is repeated for every pattern dependency relationship in the design. For large systems, the Pattern-Level with Interfaces diagrams will be developed for each subsystem or framework defined in the analysis phase.

Interface Relationships

Since pattern interfaces could be interface classes or interface operations, we expect various types of pattern interface dependencies. Interface dependencies between pattern instances can be one of four types: class/class dependencies, class/operation dependencies, operation/class dependencies, and operation/operation dependencies. Such dependencies will be further traced in the design refinement phase to relationships between internal pattern participants. The class/class dependencies are directly related to building the class diagram models of the design, and UML associations will be used to identify the relationship between classes implementing these interfaces. The operation/operation dependencies are useful in building the detailed dynamic behavior of the design, such as UML sequence diagrams that capture object interactions. The operation/class and class/operation relationships will be further used to build the class diagram models and the behavior model of the design, since they can be refined to serve both types of models.

Product

The product of this process is the Pattern-Level with Interfaces models, which are the refinement of the Pattern-Level diagrams to express the interface relationships between pattern instances.

Design Tips

When constructing the Pattern-Level with Interfaces diagram in POAD, consider the following guidelines:

- **Pick an interface.** The designer should not expend the design time trying to select and compare multiple interfaces for the same pattern. We have found that in many cases the pattern interfaces are very simple. Favoring one interface over the other may lead to the same results. For example, consider the possible interfaces of the Strategy pattern: the `Context` class or the `ContextInterface()` operation. In many cases the selection does not have any effect on the design models that follow. In this particular case the `ContextInterface()` operation itself belongs to the `Context` class, and hence the use of either is possible.

- **Look for an interface.** If the pattern documentation (in the pattern database) does not define the possible pattern interface, the designer must mine the pattern documentation for the possible interfaces. Searching for the pattern interfaces in the documentation is not an extremely difficult problem. The designer might consider studying the pattern dynamics and looking for external triggers or actions taken by the pattern participant on other design elements.
Constructing Detailed Pattern-Level Models

Purpose

The purpose of this activity is to reveal the internal design structure of the pattern instances used in the Pattern-Level with Interface diagrams. This internal structure is used in building the class diagram of the application.

Process

Explore the Pattern Structure

POAD is a structure-based composition approach when compared with the other composition approaches that we discussed in Chapter 3. In POAD we use the internal class diagram of each pattern to construct the application class diagram. Therefore, we are interested in the class diagram model of the selected patterns.

The class diagram model for each constructional design pattern is usually an essential part of the pattern documentation template. It is expected to be a part of the pattern specification that is stored in the pattern database. Referring to the discussion in Chapter 4 about the types of patterns used in POAD, we recall that the class diagram model is an essential requirement to qualify a design pattern to be considered a constructional design pattern—the types of patterns used in POAD.

In this activity the designer reveals and studies the internal class diagram of the patterns used in the Pattern-Level with Interfaces diagram. In an integrated development environment (IDE) that supports POAD, this activity is usually supported in an automated fashion. For example, the designer is given the feature to reveal and hides the class diagram model of each pattern. Figure 9-4 illustrates the Detailed Pattern-Level diagram for the simple example that we discussed in Figure 9-3. In this model the representation of the pattern instances using UML packages and the pattern interfaces using UML interfaces are still used. In addition, the internal class diagram of the pattern is revealed inside the package boundaries. For example, the Context, Strategy, ConcreteStrategyA, and ConcreteStrategyB classes are the internal participants of the FeedforwardStrategy pattern instance.

Figure 9-4. A simple example for the Detailed Pattern-Level diagram.
The designer then studies the pattern documentation, specifically the participants in the class diagram and the roles they play in solving the design problem. The knowledge gained by the designer at this activity is useful for constructing the overall application class diagram, as we will discuss in Chapter 10.

**Dynamic Aspects Are Also Important**

The dynamic aspects of a pattern help the designer understand the details of how the pattern participants communicate with each other. While the documentation of the dynamic aspects of the pattern does not play a direct role in modeling the application design in POAD, it plays an important inherent role in familiarizing the designer with the details of the class diagram model for each pattern.

In POAD the modeling process focuses on using the class diagrams of each pattern. A class diagram is the modeling artifact that is closest to the implementation phase. Classes, as modeling constructs, have one-to-one mapping with classes at the coding level. This is because every OO programming language supports classes as code constructs. However, scenarios, interactions, or object diagrams are used to analyze the runtime behavior. Although they do not have a one-to-one mapping with code constructs, they help the developer in implementing the methods of each class and in coding the method invocations and the execution sequence.

As part of this activity, the designer studies the dynamics between the pattern participant and the roles played by each participant. This will help the designer at the next phase (design refinement phase) in instantiating the internal details of the pattern to solve the particular
Realization of the Interface

As part of this activity, we identify which elements of the pattern internals are exposed as interfaces. Each pattern has a set of interfaces, which could be interface operations or interface classes. The designer identifies parts of the internal design that realize these interfaces. Every interface in the Pattern-Level with Interfaces should be traced to a participant in the pattern internals. In an IDE environment this could be made possible using a connectivity mechanism to show the connection between the interfaces and the pattern internals. This connectivity mechanism could be as simple as a UML realization relationship between the interface and the classes in the internal class diagram model.

Product

The product of this phase is the Detailed Pattern-Level diagram, which is used as input to the design-refinement phase.

Design Tips

When constructing the Detailed Pattern-Level diagram in POAD, consider the following guidelines:

- **Study the solution.** The designer should study the detailed solution provided in the documentation of the selected patterns. Both the structural aspects and the dynamic aspects are important. The structural aspects are used to build the necessary design models. The behavioral aspects help in understanding how the pattern participants collaborate to provide the solution to the problem.

- **Don’t instantiate the internals of the pattern.** In this activity, the designer should focus on understanding the solution structure and dynamics inside a pattern. It is early to try to instantiate the internal design of the pattern with application-specific details. For instance, the designer should use the class and method names specified in the pattern documentation and should avoid giving them specific application names. The instantiation of the pattern details to solve the application-specific problem is an activity in the design-refinement phase.
Summary

This chapter described the design process in POAD. The activities within the design phase serve two purposes. The first purpose is to develop the POAD design models that capture the application design as a composition of patterns at three levels of abstractions. The second purpose is to encourage the designer to obtain the knowledge about the existing solution provided by each pattern used in the design.

During the design phase, the designer produces three types of design models: the Pattern-Level diagram, the Pattern-Level with Interfaces diagram, and the Detailed Pattern-Level diagram. The three model diagrams provide three abstraction layers for the application design. These model diagrams possess several good modeling properties:

1. **Traceability**: constructs in each design model are traceable up or down the abstraction layers to other model views.

2. **Refinement**: each design model refines the models in the higher abstraction layer by adding or revealing design-specific details.

3. **Hierarchy**: some model constructs encapsulate other model elements.

Chapter 5 provided more detailed discussion about the three design models.

The models produced during the design phase are then used by the designer to develop the application class diagram during the design-refinement phase, as will be discussed in Chapter 10.
Chapter 10. Design-Refinement Phase

Overview

Instantiating Pattern Internals

Developing the Initial Class Diagram

Design Optimization

Using POAD for Developing Frameworks

Summary
Overview

In this chapter we discuss the design-refinement phase of the POAD methodology. During the design-refinement phase, we create the class diagram of the application using the class diagrams of the building block patterns. The inputs to this phase are the detailed pattern-level diagrams, which are produced from the design phase. These design diagrams are used to model the application as a collection of patterns and their interconnections and relationships. During the design process, the designer has already gained significant understanding of the internal details of the selected patterns. This knowledge will be useful in all the activities in the design-refinement phase.

The design-refinement phase has three main activities:

- **Instantiating the pattern internals**, in which we add application-specific nature to the internal design of the patterns.
- **Developing class diagrams**, in which we construct the initial class-diagram design of the application.
- **Optimizing the design**, in which we optimize the class diagrams of the application by overlapping participants of the pattern instances.

The purpose of this phase is to create the class diagram of the application using the class diagrams of the patterns and the domain-specific knowledge of the application. Starting from a set of Detailed Pattern-Level diagrams developed earlier in the design phase, the deliverables of the design refinement phase include:

- The application-specific instance of each of the design patterns used, which represents the pattern's original design model using application-specific details and naming conventions.
- The initial class diagram models of the application design, which represent the application's design diagram model as a loose assembly of patterns.
- The optimized class diagram models for the application, which represent a more dense and profound class diagram model for the application. These will represent the class diagram models for the application to be used in the detailed design and implementation phases.

As we discussed in Chapter 5, all the design models produced at this phase should be preserved. A traceability mechanism would allow the designer to traverse the design models and trace pattern participants from one level to another (see Chapter 16).

**Figure 10-1** illustrates the overall design-refinement phase. The legend used in this figure is the same as the one we used in Chapter 7.

**Figure 10-1. The POAD design-refinement process.**
In the subsequent sections we elaborate on each of the three main activities in the design process: instantiating the pattern’s internal design, constructing the initial class diagram models for the application, and optimizing the class diagrams to produce a dense, profound design.
Instantiating Pattern Internals

Purpose

The purpose of this step is to create an application-specific instance of the patterns used in the Detailed Pattern-Level diagrams.

Process

As we discussed in Chapter 9, the designer creates an instance for each selected pattern. Instantiation of a design pattern means turning it into a tangible artifact. The instantiation process involves describing the patterns and their constituents in an application-specific context. The first part of the instantiation activity is already completed during the design phase when the analyst selects an application-specific name for the pattern instance. The second part is related to instantiating the pattern internals, which is the subject of the instantiation activity in this phase.

Specialization (Generic to Application-Specific)

The design model of a pattern (in terms of a class diagram) provides a generic solution to a common problem. The pattern's class diagram model does not provide any application-specific details because it is abstracted to be reusable in multiple applications. For example, the names of the internal constituents of a pattern's class diagram are general; these names are suitable for any application but do not reflect the application-specific problem to be solved. At the design-refinement phases this generic design is specialized according to the specific application we are designing. This specialization activity includes renaming of classes, methods, and relationships and adding domain-specific details. This activity is usually referred to as specialization of a generic design to suit a specific application.

We start by creating application-specific names for the pattern internals. This step involves

- Revealing the internal class diagram model for each design pattern in the Detailed Pattern-Level diagrams.
- Renaming the internal classes of each pattern according to the application design environment. We choose names for pattern participants that are meaningful in the application context.
- Giving application-specific names to the methods and operations in each pattern class.

Let's consider a concrete example of using POAD to develop the application design of a feedback control system. In this application we assume that readings from the plant are processed and then compared to the user input (preset values for instance). The difference between the processed readings (feedback data) and the preset input is then used to trigger a control action on the plant. The designer has chosen to use an ErrorObserver pattern instance of type Observer pattern to observe the changes in the feedback data, compare it with the preset values, and trigger the control action.

The class diagram design of the ErrorObserver pattern is shown in Figure 10-2(a). In the Observer pattern, the abstract Subject class represents the abstract interface for the component to be observed, and the ConcreteSubjects classes represent concrete implementation
of the abstract subject interface. For this application, the subject we are actually observing is the feedback data, which is the postprocessed reading from the plant. Therefore, the designer might consider renaming the ConcreteSubject class to FeedbackSubject, as shown in Figure 10-2(b). Also, the abstract Observer class represents the abstract interface for the component to observe the changes in the feedback data, and the ConcreteObserver classes represents concrete implementation of the abstract observer interface. For this application, the observer is an error-processing class that compares the feedback data to the preset input. Therefore, the designer might consider renaming the ConcreteObserver class to ErrorObserver or Error class, as shown in Figure 10-2(b). Note that Figure 10-2(b) is not the complete instantiation of Figure 10-2(a); we just used a simple example of instantiating the ConcreteSubject and ConcreteObserver classes for illustration. Further details about this application are discussed in Chapter 11.

![Figure 10-2](image)

**Figure 10-2.** Instantiating the internals of the ErrorObserver instance in the feedback application: (a) the original instance, (b) the application-specific instance.

**Instantiation Through Subclassing and Realization**

Inheritance is a common mechanism used in the design of the pattern's class diagram. Inheritance allows the pattern designer to distinguish fixed parts common to all pattern instances and flexible parts to be modified during instantiation of the pattern in given applications. Therefore, we find that most pattern class diagrams have one or more inheritance hierarchy. The class diagram design of a pattern often consists of abstract and concrete classes. Abstract classes can provide only interfaces and hence become pure abstract classes or interfaces. They can also be used to provide default implementation for some of the methods that generally are used by any subclass.

During the instantiation activity of a pattern, the designer determines which abstract classes should be implemented and the concrete subclasses for each. When the abstract class is pure (an interface), the designer hooks up an interface implementation class that realizes the interface; this is called *instantiation by realization*. When the abstract class has default implementation, the designer hooks up an implementation class and determines which methods to use from the abstract class and which methods to override; this is called *instantiation by subclassing*. These two techniques are well known in the OO community and are used in POAD to instantiate the internal design of each pattern instance. Instantiation by these two techniques becomes more like extension, and not renaming.

**Keeping the History of the Participants’ Names**

The renaming process used in POAD is not simply an editing process. In editing, the designer loses the original name of a design artifact when a new name is given to it. For example, if we use editing for Figure 10-2, the SensorSubject class in Figure 10-2(a) replaces the original name, ConcreteSubject, in Figure 10-2(a). Hence, the design in Figure 10-2(a) is lost because the designer now sees the instance
Renaming in POAD involves more than editing. In POAD both the original names and the new names of the design artifacts are saved. A development environment used for POAD, such as an IDE, should keep track of the history of changes made to the pattern internals’ names. At a minimum, the models underlying POAD should save the original and the latest application-given names for all pattern constituents.

Tracking the history of changes made to the pattern is an important process for the POAD technique. It is this history tracking that enables traceability between various design models. For example, if we assume that a system without memory is used, we have no way of linking application-specific classes or methods to their original names in the pattern’s class diagram model. In many design methodologies that use patterns, this link is kept in the designer's mind and is not captured in the design models. In POAD we explicitly emphasize this feature. A development environment supporting POAD should implement this feature. For the simple example above, the designer can ask at the design level given in Figure 10-2(b) what role is played by the FeedbackSubject class. The IDE should support a tracing mechanism that allows an answer to be a ConcreteSubject class in the ErrorObserver instance of type Observer pattern.

**Concretization**

Concretization is the process of bringing abstract design to a more concrete form [Keller & Schauer 1998]. At a high abstraction level, the designer understands the problem, its abstract solutions, and various tradeoffs. At a more concrete level, we identify tactical design decisions and choose between tradeoffs.

For example, consider the Observer pattern. The design of this pattern has several variances; each variant is used to address specific force and hence has different consequences when applied. One of the design choices in the Observer pattern is to select between the pull or push modes. The push mode is used to submit the data changed in the Subject by attaching it to the notification sent to the registered Observer classes. In the pull mode, the Subject sends the notification only, and the Observer pulls the changed data (state of the Subject) later. The designer then selects between these solutions according to the application under consideration.

We note that concretization and specialization are different. Whereas specialization is concerned with turning generic solutions into application-specific ones, concretization is concerned with turning abstract designs into concrete ones by selecting among possible design alternatives.

**The Scope**

A constructional design pattern is used in POAD as a higher level of abstraction than a class. It is the building block that encapsulates other design elements (classes). The earlier design models in POAD focus more on the design of the overall application as a composition of patterns. In this phase we have changed the scope; the designer now focuses on the internal design of each pattern. This change in focus is the natural result of following the analysis and design lifecycle and is directed toward the preparation for implementation and design details.

**Product**

The product of this phase is the application-specific Detailed Pattern-Level design diagrams. These diagrams represent the complete application-specific instantiation of the patterns that were originally selected to design the application.

**Design-Refinement Tips**
When instantiating the patterns in POAD, consider the following guidelines:

- **Renaming is not editing.** We emphasize again that the designer should use tools that track the history of changes made to pattern participants. Most traditional modeling tools for OO analysis and design do not provide this feature, as we will discuss in Chapter 16.

- **Develop separate diagrams for each pattern instance.** For each pattern instance in the Detailed Pattern-Level diagrams, the designer can develop the application-specific instance in a separate diagram. This makes the design easier to handle and allows the designer to focus on instantiating the internals of the pattern with less focus on pattern interconnection and communication.

- **Abstract classes may keep the same names.** Some designers prefer to keep the original name of the abstract classes. Abstract classes are usually used inside the pattern design model to represent interfaces, and hence, it is common to provide an application-specific implementation that complies with the interface rather than change the interface abstract names to application-specific names. Inheritance is a common technique used in designing patterns to distinguish interfaces and concrete implementations.
Developing the Initial Class Diagram

Purpose

The purpose of this step is to develop a class diagram of the application using the class diagrams of the pattern instances.

Process

In this activity, we use the Detailed Pattern-Level diagrams developed from the design phase, the pattern interfaces, and the instantiated details of the pattern internals to construct a UML class diagram for the application. We produce a class diagram that is an initial step toward developing the static design model of the application.

Revealing the Instantiated Pattern Internals

In the previous activity the designer developed the application-specific pattern instances for all the patterns used in the application design. Hence, at this stage, the designer has an application-specific class diagram model for each pattern instance. The Detailed Pattern-Level diagrams developed during the design phase are now considered application-specific Detailed Pattern-Level diagrams. This is because the internal design of each pattern is now tailored for the application at hand. These diagrams have all the application-specific names for the classes and methods of each pattern instance, and are used as the basis for developing the initial UML class diagram.

Tracing the Pattern Interfaces to Internal Realization

Recall that the Detailed Pattern-Level diagrams capture the relationships between the pattern instances through the dependency relationships between the pattern interfaces. To be able to hook the internal design of a pattern instance with the internal design of another pattern instance, the designer should first trace each pattern interface to the internal pattern participant that implements or defines that interface, and then trace the relationship between interfaces to relationships between pattern internals. Figure 10-3 illustrates a symbolic diagram for these two steps.

Figure 10-3. Tracing interfaces to internal implementations.
Each pattern interface is realized by one of the internal pattern participants. This realization relationship provides another traceability feature in POAD where the interface is traced back to the pattern element implementing or defining it. This traceability mechanism can be easily supported by an IDE using a UML realization or dependency relationship.

For example, consider the FeedforwardStrategy and the ErrorObserver pattern instances of the feedback control application discussed in the previous chapter. After instantiating the internal details of each of those pattern instances (not included here), the designer traces the interfaces to the internal application-specific classes in the application-specific Detailed Pattern-Level diagram. The FeedforwardStrategy has an interface class called Context, and the ErrorObserver has an interface operation called Update(). The Context interface class is realized using the Controller class, which is an internal class in the FeedforwardStrategy pattern instance. The Update() interface is realized using the ErrorObserver class, which is an internal class in the ErrorObserver pattern instance. Figure 10-4 illustrates the application-specific Detailed Pattern-Level diagram for these two patterns. A dotted line is used to hook the interface with the internal design of the pattern instance. The notation to be used is dependent on the IDE.

**Figure 10-4.** Tracing interface to pattern internals—an example from the feedback framework.
To develop a class diagram from the application-specific Detailed Pattern-Level diagrams, we have to trace all relationships between pattern interfaces to class relationships between participants inside the pattern. At this stage, the designer has already defined the relationships between the pattern interfaces during the design phase. Those relationships are still represented in the domain-specific Detailed Pattern-Level diagram. For example, Figure 10-4 illustrates the relationship between the Update() interface of the ErrorObserver pattern instance and the Context interface of the FeedforwardStrategy pattern instance. The designer also defined which pattern internals realize the pattern interface. For example, in Figure 10-4 the Update() interface is traced to the ErrorObserver class in the ErrorObserver pattern instance, and the Context interface is traced to Controller class in the FeedforwardStrategy pattern instance. The designer now uses the relationship between the pattern interfaces and the traceability between the interface and the pattern internal to establish a relationship between the internals of the two pattern instances (the Controller class and the ErrorObserver class).

Since pattern interfaces could be interface operations or interface methods, the designer is left with various types of interface relationships:

- **Class/class relationships.** Each of the two interface classes is traced to an internal class of the two interfacing patterns. These internal classes have a class relationship.

- **Class/operation relationships.** An interface operation is traced to the class that implements it, and the interface class is traced to an internal class of the pattern. Then a class relationship is established between these two internal classes.

- **Operation/operation relationships.** Each interface operation is traced to an internal class of the two interfacing patterns. These internal classes will have a class relationship.

For the example in Figure 10-4, the Update() interface operation is defined in the ErrorObserver class, and the Context interface class is defined by the Controller class. The class diagram for these two patterns is illustrated in Figure 10-5.

**Figure 10-5. The initial class diagram for two pattern instances.**
Similarly, the designer traces all interface relationships between all pattern instances to relationships between the classes in the internal class diagram model of those pattern instances. As a result, the application design is now modeled as a UML class diagram.

Product

The product of this activity is the initial UML class diagram for the application design. These class diagrams are neither dense nor profound, and they represent a loose assembly of the internal design of the pattern instances.

Design-Refinement Tips

When developing the initial class diagram, consider the following guidelines:

- **Should we perform detailed design yet?** During this activity, we create the initial class diagram design of the application. This activity is not meant to develop the detailed design of the application. For example, the designer might want to add more methods or application-specific variables to the internal classes of each pattern. The design-refinement phase is different from the application’s detailed design phase. The detailed design phase is the next phase in the lifecycle and uses traditional OO methods.

- **Keep the interface details.** During the refinement of interface relationships between pattern instances, some design details might be lost. For example, consider the case in which the pattern’s interface relationship is between an interface operation and an interface class. During the refinement process, this relationship is translated into relationships between classes inside the pattern instances. Hence, the knowledge of which operation in the class is interacting with the other class is lost in the initial class diagram. Though this knowledge is still preserved in the Detailed Pattern-Level diagram, the designer might want to make it more explicit in the initial class diagram. In such cases, the designer can write a comment using UML notes and attach it to the UML class representation in the class diagram model.
Design Optimization

Purpose

The purpose of this step is to optimize the initial UML class diagrams that we developed in earlier activities in the design-refinement phase. The optimized class diagram is a more dense and profound version that is ready for detailed design and implementation.

Process

The initial class diagram obtained from gluing patterns together at the high-level design is neither dense nor profound. This initial class diagram is a loose assembly of pattern instances. As a result, the design might contain replicated abstract classes or many classes with trivial responsibilities. In this activity, we use reduction, merging, and grouping steps to optimize the design.

Reduction

During the design process, the designer might create multiple instances of the same pattern type. The internal design of a pattern usually contains abstract and concrete classes. The abstract classes can sometimes be pure abstract classes that carry no implementation (also referred to as interface classes). The abstract classes are used inside the pattern’s design model to provide default implementation for some methods that are shared with all concrete pattern classes implementing the abstract class. The pure abstract classes (interfaces) are used to define the common interface that is realized (implemented) by the pattern’s concrete classes. When creating application-specific instances of the pattern, the designer usually keeps the abstract classes or interfaces unchanged and adds his own concrete implementation by subclassing the abstract class or by realizing the interface using an application-specific class. Instantiation through subclassing and instantiation through realization are very popular mechanisms in instantiating a design pattern.

Since the same pattern type can be instantiated in more than one pattern instance, the abstract classes in that pattern type will be instantiated more than once. If the designer does not change any elements in these abstract classes, the initial design will contain replicated abstract classes coming from several instances of the same pattern type. For example, consider the feedback control application that we discussed earlier. The designer uses the two pattern instances SensorObserver and ErrorObserver of type Observer pattern as a mechanism for monitoring the changes in the plant and monitoring the changes in the feedback data. The Observer pattern has two abstract classes, the Observer and Subject abstract classes. Thus, the design of the feedback application will contain two abstract classes named Observer and two abstract classes named Subject.

Reduction is the process by which the designer removes the replicated abstract classes that result from using multiple instances of the same pattern type. An abstract class should appear once and be subclassed according to concrete implementations. Similarly, interface classes should appear once in the design, and different implementations are then realized in concrete, application-specific classes. The designer then uses one class to represent the replicated abstract class. For example, the designer would use one abstract class called Observer to represent both Observer abstract classes in the SensorObserver and ErrorObserver pattern instances. The same reduction could be used for the Subject abstract class. Figure 10-6 illustrates the design model of the SensorObserver pattern instance, the design model for the ErrorObserver pattern instance, and the design after the reduction activity and removal of the replicated abstract classes.

Figure 10-6. Reduction process.
This reduction activity is not usually feasible in every application design. There is one important precondition to using one abstract class instead of two: the two classes have to be similar. This requires the designer to assess the similarities and differences in the abstract classes or interfaces in the two pattern instances. In the case of interfaces the common interface class has to provide the same interface signature for both pattern instances. In the case of abstract classes the common abstract class has to provide not only the same interfaces but also the default implementation for the similar interfaces should be the same for both pattern instances. These replicated classes can be removed only if they satisfy these conditions. Therefore, the replicated classes are eliminated, and only one common version of the abstract classes (interface) is used. For example, the abstract class `Subject` in the feedback control application provides interfaces to hook observers and to notify the observers whenever there is a change in the subject. If the designer does not change any element in this class for the SensorObserver or ErrorObserver pattern instances, then we can use the same class to represent both abstract classes. However, if the designer changes the implementation of the `notify()` method in one abstract class such that it is a pull mode for the SensorObserver instance and a push mode for the ErrorObserver instance, then the two `Subject` classes are different, and we cannot use one abstract class to represent both.

The complexity of the design is reduced by eliminating replicated abstract classes or interfaces. The development environment used in POAD could provide the designer with replicated abstract classes that could be merged by tracking down the pattern instance creation and pattern instances that have the same type.

**Merging and Grouping**

As we discussed in Chapter 7, many software designers are against developing designs that are loose assembled patterns because the design structure of patterns might contain classes with trivial responsibilities. Some design patterns contain classes that are just used to forward a message to another class inside the pattern structure. Such classes are not meant for as-is use. These classes should carry
other functionalities in the application besides the functionality that they provide as internal participants in the internal design structure of the pattern.

For example, consider the internal design structure of the Strategy pattern, shown in Figure 10-7. The Context class in this design represents the interface for the strategy that is used inside the pattern. This interface class is meant to be part of the application design to which a concrete strategy will be hooked and will be responsible for invoking the strategy at runtime. Therefore, the Context class would play other roles in the application design that is not just the interface to the encapsulated strategy.

**Figure 10-7. The context class in a strategy pattern will play other roles in the application design.**

Instead of using simple classes with trivial responsibilities, the designer at this stage would consider using one class to implement the responsibilities and roles of participants from more than one pattern instance. Merging (also called grouping) is the process by which the designer defines one class to implement the roles of participants from more than one pattern instance.

Within the merging activity are two important steps:

- Identifying which classes to merge
- Defining how to merge the classes

In order to identify which classes to merge, the designer must study the internal design of the pattern instances as well as the relationship between them. Using the internal design of a pattern instance, the designer can identify classes—for example, the Context class in the Strategy pattern—that carry out simple responsibilities. To judge whether a class has trivial responsibilities or not, the designer studies the role played by each participant, as documented in the pattern template. The designer should also be able to assign application-specific responsibilities to those participants according to the application at hand. Based on information from the pattern documentation as well as application-specific information, the designer determines whether a pattern participant is a candidate for merge with other application design classes.

Another method for identifying which participants are candidates for the merge operation is through studying the pattern relationships and interfaces. The relationship between two pattern interfaces is mapped (as discussed earlier) to relationships between participants from two pattern instances. Those participants are candidates for merging. The designer then studies the functionalities assigned to these interfaces. Classes that perform highly related responsibilities are identified. Instead of implementing a primitive function in each class, the designer merges these classes into one class, which will carry out the responsibilities of all the merged participants.
There are several techniques to merge the implementation and design of two classes:

- **MixIn inheritance.** In this technique we use one class that inherits from both classes to be merged. The new class may redefine a method to realize the effect of the merger. Naming collision may occur, and hence the modeling environment should be able to support renaming of methods. With MixIns, a new class is added to the model, and the original classes remain in the design. Thus, when using MixIns, more classes are added, and the original classes—with their trivial responsibilities—still exist. As a result, this method is not used in POAD because no optimization is achieved.

- **Restructuring of inheritance hierarchy.** In this case the inheritance tree is readjusted to make one of the classes inherit from the other. Such operations are usually very difficult and may cause design errors. In addition, the design is not actually optimized, since the two classes still exist in the application design.

- **Hard merge two classes into one.** In this case, the designer merges the two classes to produce one class that has no conflicting methods and attributes.

In POAD, we use the hard-merge operation, since we end up with one class representing both of the original classes. The two original classes are no longer used in the application design model and can be deleted. When we merge two classes that have no association with each other (i.e., there is no class relationship between them), the new class will have relationships with other application classes, which are the collection of relationships between each of the original classes and the application classes. When we merge two classes that have an association relationship, the association between the two original classes is retained in the merged class as calls between internal class methods.

To illustrate the merge activity, consider the feedback control application in which the two pattern instances FeedforwardStrategy of type Strategy pattern and ErrorObserver of type Observer pattern are used. The two pattern instances are illustrated in Figure 10-5. As we discussed earlier, the ErrorObserver class is a concrete observer that invokes the Controller class (originally the Context in the Strategy pattern) to apply a specific control strategy on the plant once there are changes in the feedback data and the error data. The Controller class is the interface for the Strategy pattern and represents the application-specific context that invokes the attached control strategy. Since this class carries a trivial responsibility of invoking the attached concrete strategy, it is a good candidate for merge. In this application the designer might consider merging the two classes Controller and ErrorObserver into one class called ErrorObserverController, for instance. In this case the ContextInterface() method of the Controller class would be moved to the ErrorObserver class, and the Update() method of the ErrorObserver class will invoke the ContextInterface() method internal to the same ErrorObserver class. As we can see, there are no naming conflicts in this example. Figure 10-8 illustrates the optimized diagram.

**Figure 10-8. Merging activity in the feedback control framework.**

The IDE environment supporting the POAD methodology should provide the merge feature to the designer. The designer should be able to select two classes for merger, and the IDE should generate the resulting class. The merge could result in naming conflicts between the design elements within the merged classes. The IDE should provide the means to present such conflict to the designer as well as the means for providing merge solutions. The characteristics of such IDE are discussed further in Chapter 16.
Product

In this activity optimization in the class diagram is sought by removing replicated abstract classes and merging participants from several patterns. The product of this phase is an optimized class diagram for the application, which is more dense and more profound.

Design-Refinement Tips

When optimizing the application class diagram, consider the following guidelines:

- **Merge in pairs.** When the designer is merging classes, it is often easier to consider merging two classes at a time. Once two classes are merged, the result can be merged with another class, and so on.

- **Look for replicated classes.** The design might contain replicated abstract classes from several pattern instances that are not identical. If there are slight differences in the interfaces of the abstract classes, the designer can consider merging the classes and accommodating the slight differences in the resulting class.

- **Design skills.** The merging and grouping step brings the class diagram to a more dense form. However, it mainly depends on the designer's skills to identify the classes to merge and how to merge them. The designer should become familiar with the details of each pattern instance before starting the merge and reduction activities.
Using POAD for Developing Frameworks

The same POAD process that is used for designing software applications is also useful in developing pattern-oriented frameworks. Due to the generic nature of a framework, POAD users might want to consider some additional activities:

1. In the POAD requirement analysis phase of a framework, we analyze a domain of applications instead of one concrete application. The result of domain analysis is a set of conceptual components, which are commonly used in applications that are built using the framework. Design patterns can often represent these components because they are the result of refactoring several application development experiences, usually in the same domain.

2. After the design-refinement phase, the design of the framework is refactored again to separate the abstract parts from the concrete ones. Refactoring of pattern-oriented frameworks is not as tedious as refactoring traditional OO designs. This is because pattern-oriented frameworks are built from patterns that are themselves refactored designs.

3. A pattern-oriented framework can be expressed using a pattern-level view or a class diagram view. Instantiating the framework for a specific application can start from a high abstraction level of pattern diagrams or lower abstraction level of class diagrams. Figure 10-9 illustrates the instantiation alternatives of a pattern-oriented framework. The benefits and disadvantages of each instantiation approach were discussed in Chapter 7.

Figure 10-9. Instantiation of pattern-oriented frameworks.
Summary

This chapter described the design-refinement process in POAD methodology. The activities within the design-refinement phase are focused on generating the class diagram model for the application. We start by instantiating the pattern's internal design using application-specific details. An initial class diagram is then created by removing the pattern boundaries and tracing pattern interfaces to class relationships between the pattern instance's internal classes. Finally, the initial class diagram is optimized using a reduction and merging mechanism to obtain a more dense and profound class diagram. The output from this phase is the application class diagram models that will be used in the detailed design and implementation phase.

The detailed design phase is different from the design-refinement phase. The objective of design refinement is to develop a class diagram model of the application. The detailed design phase focuses more on adding application details, such as more class methods and class attributes, by using traditional OO analysis and design methods and models.

Starting with the optimized class diagram and the pattern-oriented views of the design that are developed in the POAD methodology, the designer can further analyze the system using traditional OO design models such as collaboration, interaction, and statechart models. The development process is iterative. As analysis and design proceeds, using sequence diagrams—for example, the relationships between classes—could change. These changes have to be reflected in the Pattern-Level diagrams. A development environment is required to support the POAD steps and maintain the consistency between design models. Further detailed design steps follow traditional OO techniques by defining details and behavioral aspects of classes and other design artifacts.
Part IV: Case Studies

To illustrate the applicability of the POAD methodology to develop pattern-oriented designs and frameworks, we illustrate examples of applying the methodology to four case studies.

Chapter 11 follows POAD steps to develop a pattern-oriented design framework for feedback control systems. The framework is generic and is easily instantiable in developing application-specific control systems. We include sample code in Java for a complete application.

Chapter 12 follows POAD steps to develop a pattern-oriented design for the domain of simulation of waiting queues. In this application we deal with customers lining up for service from one or more service stations, such as supermarket checkout counters or a self-serve car wash. We also include sample code in Java for a complete application.

Chapter 13 follows POAD steps to develop a pattern-oriented design for the domain of digital content processing. This application is used to read, process, and manipulate digital content where heterogeneous source and delivery channels are supported and metadata information is extracted while processing the digital content.

Chapter 14 follows POAD steps to develop a pattern-oriented design for distributed medical informatics systems based on the Digital Imaging and Communication in Medicine (DICOM) standard.
Chapter 11. Feedback Control Systems

Control systems are widely popular in real-world applications. They are often implemented in many software applications, whether to control an external environment or to control other software components in a system. In this chapter we use the POAD process and models to develop a reusable pattern-oriented design framework for feedback control systems.

Design patterns have been deployed in the design of many domain-specific applications that have feedback properties. For instance, patterns are used to develop the manufacturing system discussed by Schmid in [Schmid 1995]. The development of such a framework starts by using traditional OO constructs such as classes and objects. Design patterns are then used to develop a generic and flexible framework by refactoring the existing design. Through the use of patterns, the design framework becomes generic enough to use in many automated manufacturing systems like assembly lines. Another experience of using patterns in developing applications for manufacturing systems is illustrated by Bosch in "An Object-Oriented Framework for Measurement Systems" (1998a). These experiences and others are practical examples of using patterns as fundamental elements in the design of software applications for manufacturing systems.

Feedback systems are commonly modeled using block diagrams. The design framework that we develop in this chapter is based on design patterns as building constructs. The framework is documented at various design levels using POAD models and is reusable as an initial phase in designing feedback control applications. To develop a pattern-oriented design for feedback control systems, we follow the POAD process outlines, as discussed in Chapter 7, and the detailed process described in chapters 8, 9, and 10.

Recall the three development phases outlined in Chapter 7: analysis, design, and design refinement. Each phase contains activities (steps), as discussed in chapters 8, 9, and 10. In the following sections we discuss the application of the POAD activities to develop the framework for feedback control systems. There is no absolute need to follow a pure waterfall model; the designer initially follows these activities and can always iterate on them as long as the models developed from each step are documented and traceable to the outcomes of other activities.
POAD Analysis for the Feedback Control Framework

Requirements Analysis

A closed-loop control system can be decomposed into components based on independent responsibilities. To implement a feedback control system, the specification and description of the system configuration and its components must be put into a form amenable to analysis and design. Three basic representations (models) of components and systems are used extensively in the study of control systems: mathematical models, block diagrams, and signal-flow graphs. Referring to control literature [e.g., Distefano et al. 1990], the generic block diagram of feedback systems represents an initial architecture documentation to start with. A feedback control system is one in which the control action is dependent on the output. Feedback is the property of a closed-loop system that permits a measured output to be compared to the system's input so that the appropriate control action may be performed as some function of the output and the input. Figure 11-1 illustrates the block diagram that is often used to describe a feedback control system.

Figure 11-1. Block diagram for a feedback control system.

Many practical reactive systems encompass software control applications in their implementation. The portion of a system to be controlled is usually called the plant. A plant is to be accurately controlled through feedback operation. An output variable is adjusted as required by the error signal. The error signal is the difference between the system response as measured by the feedback element and the reference signal, which represents the desired system response. Generally, a controller is required to process the error signal such that a certain control strategy will be applied. We are not concerned here with the theoretical analysis techniques of a feedback system; instead, we concentrate on the pattern-oriented design of the feedback system in a reusable form that is easy to implement and use.

To analyze a feedback system, we may use a UML use case diagram. Figure 11-2 illustrates a use case diagram for the feedback system. In this diagram, we identified three actors interacting with the system: the user who configures the system and provides the input data for adjusting the plant, the sensors that measure some data from the plant, and the actuators that control the plant. We have also identified several use cases: the configure use case in which the user provides the input parameters used in controlling the plant; the monitor use case in which measurement data is collected; the control use case in which the plant is controlled by some actuators; the calculate error use case, which determines the difference between the measurements taken from the plant and the required values set by the user; and the collect data use case, which collects all statistics and measurements taken during the operation.

Figure 11-2. A use case diagram for the feedback system.
Further analysis can proceed at this level by starting the development of interaction diagrams for the identified use case. Interaction diagrams help in defining the components and their interactions, and hence we are able to determine the logical components and their functionalities. However, for this case study, the use cases are simple, and it is easy to determine a set of logical components.

Using the generic block diagram of a closed-loop control system and the use case diagrams, the system is decomposed into the following components:

- A *feedforward* component, which handles the error data and applies a control algorithm to the plant.
- A *feedback* component, which takes measurement data from the plant, processes it, and provides the feedback data.
- An *error calculation* component, which compares the input and feedback data and produces the error.
- The *plant*, which is an external component controlled by the system.

At the end of this activity, the system is decomposed into a set of components, and the functionalities and responsibilities of each component are identified. The description provided above is succinct but sufficient to proceed to the next development activities. Now we proceed to the process of selecting suitable patterns to implement the functionalities of these conceptual components.

### Pattern Selection

We analyze the responsibilities and the functionalities of each component and identify candidate patterns that could provide a design solution for each component. In this step, we consider the design problem that we want to solve and match it to the solution provided by general-purpose design patterns, as those documented in [Gamma et al. 1995; Buschmann et al. 1996; PLoPD, PLoPD2, PLoPD3, PLoPD4]:

1. The *feedforward* component implements a control strategy. The design should provide flexibility in the selection and deployment of various control strategies with minimal impact on other components in the system. For example, the feedforward component should provide the same interface to the rest of the components in the system, while the framework can provide the flexibility to plug in and take out different control strategies. If we consider this as the design problem that we want to solve and search for patterns whose intent is to solve similar problems, we find that a *Strategy* pattern [Gamma et al. 1995, pp. 315] is a good candidate for this task. Recall the intent section of the Strategy pattern: “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” Hence, the
Strategy pattern solves the design problem of the feedforward component.

2. The feedback component receives measurements and applies a feedback control strategy. It feeds the results from comparing (according to a specific strategy) the feedback data with the input data to the error calculation component. The measurement unit observes and measures data from the plant and feeds it to the feedback branch. A mechanism is required to deliver measurements to a feedback controller. Measurements can be delivered to the feedback controller using the Observer pattern [Gamma et al. 1995, pp. 293]. Recall the intent section of the Observer pattern: “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” Thus we can use the Observer pattern to loosen the dependency between the objects doing the plant observation and those actually doing the feedback control operation. Measurement data is fed to the feedback control strategy, which—similar to the feedforward component—applies a specific control strategy. The design should provide the capability to plug in and take out different feedback control strategies. This can be implemented using another Strategy pattern.

3. In the error calculation component, the feedback controller notifies the error calculation unit with the feedback data. The feedback controller can be viewed as the subject that notifies the error calculator with changes in the feedback data. Error calculation is done at the moment feedback data becomes available; this data is compared with the input data. Thus, an Observer pattern can implement this behavior.

4. If we examine various data manipulated in the feedback system, we find that the system manipulates measurement data that is measured from the plant; feedback data that is the result of processing the measured data by the feedback element; and error data that is the result of processing the feedback data and the input data. Data of different types need to be exchanged between the framework components. We can use a Blackboard pattern (a modified version of the blackboard patterns in [Rogers 1997; Buschmann et al. 1996]) for managing the system repository.

To summarize, in this step a set of patterns is selected to fulfill the responsibilities identified for each conceptual component—that is, those that we identify in the requirement analysis activity. In choosing these patterns, we consider how the pattern solves the design problem and the intent of the pattern. A Strategy pattern is selected for the feedforward component, an Observer and a Strategy pattern are selected for the feedback component, an Observer pattern is selected for the error calculation component, and a Blackboard pattern is selected as the system repository. In this simple example, it is obvious which patterns can be used. In other complex examples, the analyst could use UML use cases and sequence diagrams to understand the functionality required by each component.
POAD Design for the Feedback Control Framework

Constructing Pattern-Level Diagrams

In this step we create instances of the selected patterns and identify the relationships between these instances. As a result, a Pattern-Level diagram of the system is developed.

First, we create pattern instances. Recall that in this step we just give domain-specific names to abstract patterns types. In the previous step, we have chosen two Strategy patterns: one in the feedforward component and the other in the feedback component. Thus, we use the instances `FeedforwardStrategy` and `FeedbackStrategy` of type Strategy pattern in the design of the feedforward and feedback components respectively. We have also selected two Observer patterns: one for the feedback component and the other for the error calculation component. Thus, we use a `FeedbackObserver` instance of type Observer pattern to observe and measure data from the plant and an `ErrorObserver` instance of type Observer pattern to calculate the error. We use a `Blackboard` instance of type Blackboard pattern to manage the system data repository.

Second, we define dependency relationships between pattern instances. The `FeedbackObserver` uses the `FeedbackStrategy` to apply a feedback control algorithm, which in turn uses the `ErrorObserver` to calculate the error. The `ErrorObserver` uses the `FeedforwardStrategy` to apply a forward control algorithm. The `Blackboard` is used by all patterns to store and retrieve data.

Finally, we use the pattern instances and their relationships to construct the Pattern-Level diagram, as shown in Figure 11-3. We use UML stereotypes to show the type of the pattern instance.

**Figure 11-3. A Pattern-Level diagram for feedback control systems.**
The product of this step is the Pattern-Level diagram of the framework. It describes the architecture of a feedback system using design patterns, which explains why the names *Pattern-Oriented Analysis and Design* and *Pattern Oriented Framework* are used. This diagram could be revisited at a later phase to iterate on the decisions that we made (as part of an iterative development lifecycle). For instance, during the design or design-refinement phases, we might discover that a selected pattern has limitations or impacts on other design aspects. In this case the designer revisits the Pattern-Level design diagram to choose another pattern and replaces previous choices or creates a new pattern dependency or a new uses relationship.

**Constructing the Pattern-Level with Interfaces Diagram**

In this step, we analyze the relationships between pattern instances. The dependency relationship between patterns in the pattern-level view is a conceptual, high-level dependency relationship that should be further traced to lower-level design relationships between pattern interfaces.

First, we declare interfaces for the patterns used in each Pattern-Level diagram (only one diagram for the feedback system). Interfaces for most of the patterns used in our case studies are defined in Appendix A. The Strategy pattern has the class `Context` as the interface to the encapsulated control strategy. The Observer has two interfaces that allow coordinating the subject observed with its observer. These interfaces are the `notify()` interface operation in the subject and the `update()` interface operation in the observer. The Blackboard pattern has the interfaces to get and store data in the repository; these interfaces are the `getData()` and `setData()` interface operations. Note that at this level do not consider any details related to the parameters used and their types.

We then identify the relationship between pattern interfaces by translating all dependency relationships between patterns in a Pattern-Level diagram to relationships between interface classes and/or interface operations. The product of this process is the Pattern-Level with Interfaces diagram. Figure 11-4 illustrates the Pattern-Level with Interface diagram for the feedback control framework.

**Figure 11-4. A Pattern-Level with Interfaces diagram for feedback control systems.**
Let's take an example: the relationship between the FeedbackObserver and the FeedbackStrategy pattern instances in the Pattern-Level view. The relationship between these two patterns is that the FeedbackObserver uses the FeedbackStrategy to apply a feedback control strategy whenever the measurement data is ready. The interfaces of the FeedbackObserver are the Update() and the notify() interface operations. The interface of the FeedbackStrategy is the Context interface class. Thus, the relationship between these two patterns is translated to a relationship between the Update() interface operation of the former and the Context interface class of the latter. Similarly, all pattern relationships of Figure 11-3 are translated to relationships between the pattern interfaces in Figure 11-4.

**Constructing Detailed Pattern-Level Diagrams**

To construct the Detailed Pattern-Level diagram, we express the internals (i.e., participants) of each instantiated pattern in the Pattern-Level with Interfaces diagram. Since we have used pervasive design patterns in developing the feedback control framework, their structure can be found in the literature. For example, the class diagram model for the Strategy and Observer patterns is documented in the GoF book (1995). The diagram in Figure 11-5 shows the Detailed Pattern-Level diagram for the feedback pattern-oriented framework.

*Figure 11-5. A Detailed Pattern-Level diagram for feedback control systems.*

Note that we do not make any additional design decisions in this step. With the appropriate tool support, Figure 11-5 is a direct generation from the Pattern-Level with Interfaces diagram by simply retrieving the internal class diagram model for each pattern from a pattern database.
POAD Design Refinement for the Feedback Control Framework

Instantiating Pattern Internals

In this step we add domain-specific nature to the Detailed Pattern-Level diagrams by renaming internal pattern classes according to the application domain, choosing names for pattern participants that are meaningful in the application context, and defining domain-specific names for operations in the patterns. In the following, we instantiate the pattern internals for each pattern that appears in Figure 11-5.

The FeedforwardStrategy pattern (Figure 11-6) is composed of

- **Controller.** The context of the control strategy that is configured with a concrete control strategy object through a reference to an `AbstractController` interface.
- **AbstractController.** The interface for all concrete control strategies. The `Controller` uses this interface to invoke the concrete control strategy algorithm through polymorphism invocations.
- **ControlStrategyA** and **ControlStrategyB.** These are concrete control strategies that represent various implementations of the control strategies that the designer can choose from.

![Figure 11-6. Instantiating the FeedforwardStrategy pattern.](image)

The error calculation component consists of the ErrorObserver pattern (Figure 11-7), which is composed of

- **AbstractObserver.** An updating interface for objects that are notified of changes in the subject.
- **AbstractSubject.** An interface for attaching and detaching observers. It knows about its observers that ought to be notified of a subject's change.

- **ErrorObserver.** It is a concrete observer that maintains a reference to the `FeedbackSubject`, reads the feedback data after being processed by the feedback strategy, analyzes the feedback data with respect to the reference input data, and stores the error in the blackboard. It implements the `AbstractObserver` updating interface.

- **FeedbackSubject.** It is a concrete subject that sends notification to the concrete observers of new data received from the feedback component.

Figure 11-7. Instantiating the ErrorObserver pattern.

The FeedbackObserver ([Figure 11-8](#)) is used in the feedback component and is composed of

- **AbstractObserver** and **AbstractSubject.** They play an interface role similar to that of the ErrorObserver pattern.

- **MeasurementSubject.** It receives measurement notifications from the plant and notifies its observer `FeedbackObserver`, that a measurement is ready.

- **FeedbackObserver.** When notified by changes in the plant (through the `MeasurementSubject`), it pulls the data identifier from the subject (using the pull mode of the Observer pattern) and invokes the feedback controller to process the measured data.

Figure 11-8. Instantiating the FeedbackObserver pattern.
The Blackboard ([Figure 11-9]) is composed of

- **Blackboard**. It is the interface for retrieving and storing data. For the error component, it stores error data for successive readings and provides interfaces for retrieving and storing data records.
- **DataHolder**. It is an interface to all types of data. This class is added to facilitate manipulating and referring to data in various class method signatures.
- **ErrorData**. The concrete error data record to be stored in the application repository.
- **MeasuredData**. The concrete data record measured from the plant.
- **FeedbackData**. The data after being processed by the feedback control strategy.

**Figure 11-9. Instantiating the Blackboard pattern.**

The FeedbackStrategy ([Figure 11-10]) pattern is composed of

- **Feedback**. It is the context of the feedback control strategy. It is configured with a feedback control strategy object through a reference to an FBAbstractController.
- **FBAbstractController**: It is the interface for all feedback control strategies. The Feedback uses this interface to call the feedback concrete algorithm.
- **FBControlStrategyA** and **FBControlStrategyB**. They represent concrete implementations for feedback control strategies.
The FeedbackObserver invokes the control routine of the Feedback that applies the feedback control strategy required from the component. The Feedback class interacts with the FeedbackSubject of the observer pattern in the error calculation component and invokes its notify() procedure. This establishes the link between the feedback component and the error calculation component.

We note that the participant names that are popular and frequently used in the pattern design are now domain-specific names. Two features can help the designer keep track of the patterns. First, the three model diagrams—Pattern-Level, Pattern-Level with Interfaces, and Detailed Pattern-Level—provide documentation of the framework as a composition of patterns. Second, with the appropriate tool support, the renaming process is not an editing process. In editing we simply change the names, and the old names are lost. But in the renaming process of a class, the tool support for POAD should provide a system with memory to keep the history of the changed name specifically in pattern instantiation, as discussed in Chapter 16.

Developing an Initial Class Diagram

From the Detailed Pattern-Level diagram, we use pattern interfaces and the instantiated details of pattern internals to construct a UML class diagram. This is a simple process of combining the domain-specific details discussed in the previous section with the Detailed Pattern-Level diagram. The class diagram developed at this phase is an initial step to develop the static design model of the pattern-oriented framework. We convert the class/class interface relationship between patterns by tracing each of the two class interfaces to the internal class of each pattern. These internal classes will have a UML-association relationship. We convert the class/operation interface relationship by tracing each interface operation to the class that implements that interface operation. The class is traced to the pattern’s internal class, and hence a class relationship is established between internal classes of the two interfacing patterns. We convert the operation/operation interface relationship by tracing each interface operation to the internal class of the pattern. The internal classes of the two interfacing patterns will have a class-association relationship. As a result, the design is now viewed as a UML class diagram, as illustrated in Figure 11-11.

Figure 11-11. The initial class diagram of the feedback design framework.
It can be easily recognized that the patterns are still notable in the class diagram as shown by the dotted boxes around the classes. We recall that we do not discard or swap away earlier diagrams. As part of POAD, all the models in Figures 11-3 through 11-11 are saved as analysis and design models. It is the role of a tool support to save these models and provide the necessary traceability mechanisms between them.

How About Code Generation?

In the above process, we do not generate code at each level and keep it in synchronization with other levels. The process is mostly about manipulating the design elements, such as patterns, classes, and methods, at that analysis and design level. Code generation comes later, after the refined design is complete. It is also unnecessary to generate code from the initial class diagram because in the following phases the designer will change the design of the system by merging and grouping several classes together. So, if you are eager to see code samples, wait until you have completed the following phase.

Design Optimization

The class diagram obtained from gluing patterns together at the high-level design is neither dense nor profound, because we just strung the patterns together. It has many replicated abstract classes due to using multiple instances of the same pattern. For example we used the FeedbackStrategy and the FeedforwardStrategy instances of type Strategy pattern. It also has many classes with trivial responsibilities because many classes are there for forwarding messages to internal participants of the pattern. Therefore, in this step we use reduction and grouping mechanisms to optimize the UML design diagrams obtained initially in the previous step.

Reduction
In this step the complexity of the framework is reduced by eliminating replicated abstract classes. A pattern has one or more abstract classes. Since the same pattern type is used in more than one instance, we expect to find similar abstract classes. An abstract class, in its sense, should appear once and be subclassed according to concrete implementations. From the above figures, the Observer pattern is used in the feedback component and in the error component. Thus, the classes AbstractObserver and AbstractSubject are replicated. Similarly, the abstract class AbstractController of the strategy pattern used in the feedforward and feedback components. Therefore, the replicated classes are eliminated, and only one common version of the abstract class is used.

This step is not usually applicable to all designs, because the interfaces offered by abstract classes may substantially differ, and hence we cannot merge these two abstract classes into one class. However, for the feedback control system this is feasible. In general, this is an activity that the designer might consider as part of the development process.

**Grouping**

In this step more optimization in class usage is projected by merging concrete classes together depending on their interaction and responsibilities. This step brings the framework’s class diagram to a more reduced form; however, it mainly depends on the framework designer’s skills. From Figure 11-11, we find that the classes FeedbackObserver, FeedbackSubject, and Feedback perform highly related functions, which are summarized as receiving measurement notification, applying control strategy, and notifying the error component that the feedback data is ready. Instead of implementing a primitive function in each class, we can merge these three classes into one class, FeedbackSubjectObserver, which carries out the responsibilities of the three classes.

As a result of the reduction and grouping activities, the design becomes more dense and, depending on the designer's skills, more profound. Figure 11-12 illustrates the refined class diagram of the framework.

**Figure 11-12. The refined class diagram for the feedback design framework.**
It could become difficult to identify the patterns at this level because the design is now represented in terms of domain-specific classes. This problem has always existed in many techniques that use patterns directly at the class diagram level without developing higher-level design models. POAD has one particular advantage. When applying POAD, the designer keeps all the models developed throughout the development lifecycle. These models are traceable bottom-up from the class level (Figure 11-12) to the pattern level (Figure 11-3) and top-down from the pattern level to the class level. This traceability can be automated in a design environment.

As an example of top-down traceability, we can identify the pattern participants in the above class diagram as follows:

- **FeedforwardStrategy**: Strategy is composed of the classes Controller, AbstractController, ControlStrategyA, and ControlStrategyB.
- **FeedbackObserver**: Observer is composed of the classes AbstractSubject, AbstractObserver, MeasurementSubject (a concrete subject), and FeedbackSubjectObserver (a concrete observer).
- **ErrorObserver**: Observer is composed of the classes AbstractSubject, AbstractObserver, FeedbackSubjectObserver (a concrete subject), and ErrorObserver (a concrete observer).
- **FeedbackStrategy**: Strategy is composed of the classes FeedbackSubjectObserver (the controller), AbstractController, FBControlstrategyA, and FBControlstrategyB.
- **Blackboard**: Blackboard is composed of the Blackboard, DataHolder, and the inherited data subclasses MeasuredData, FeedbackData, and ErrorData, which are information data classes used by MeasurementSubject, FeedbackSubjectObserver, and ErrorObserver.

As an example of bottom-up traceability, we find that the class FeedbackSubjectObserver is a common participant in multiple patterns:

- It is the observer for MeasurementSubject in the FeedbackObserver. It is notified by changes in the MeasuredData.
- It acts as a controller in the FeedbackStrategy that invokes a concrete control strategy to be applied on the FeedbackData.
- It acts as a subject for the ErrorObserver. It notifies the observer of changes in the FeedbackData.

The following Java code sample shows the FeedbackSubjectObserver class, which has become the integration of the above three classes coming from three different pattern instances. We specifically note the following:

1. The update() method has code that belongs to three roles: the role of an Observer in the FeedbackObserver pattern instance, the role of a Subject in the ErrorObserver pattern instance, and the role of a Context in the FeedbackStrategy pattern.
2. The concrete observers use the pull mode in which the observer pulls data from the subject by calling its getState() method after the observer has been notified that a change has occurred in the subject data.
3. To overcome the problem of multiple inheritances in Java, we defined AbstractSubject as an abstract class and AbstractObserver as an interface. FeedbackSubjectObserver implements the AbstractObserver interface to provide the observer role and extends AbstractSubject to provide the subject role. Other implementations are also possible.

```java
public class FeedbackSubjectObserver extends AbstractSubject implements AbstractObserver {
    protected AbstractController FBController;
    protected DataHolder feedbackData;
    protected Blackboard myBlackboard;

    /**
     * public void update(AbstractSubject changedSubject)
     */
```
{  
    // When notified, get the data from the measurement subject
    // Role. Playing the role of a concrete observer class in an
    // Observer
    DataHolder measuredData;
    measuredData = changedSubject.getState();

    // Role. Playing the role of a context class in a Strategy pattern
    FBController.FBApply(measuredData ,feedbackData);
    myBlackboard.setData(feedbackData);

    // Role. Playing the role of a concrete subject class in another
    // Observer
    notifyObservers();
}

/**
 * public DataHolder getState()
 * {
 *     return feedbackData;
 * }
 * public FeedbackSubjectObserver()
 * {
 *     feedbackData = new DataHolder();
 * }
 * public void setFBController(AbstractController aController)
 * {
 *     FBController = aController;
 * }
 */

/** A setter method for the storage object */
public void setBlackboard(Blackboard theBlackboard)
{
    myBlackboard = theBlackboard;
}
}

Through the reduction and grouping steps, we combine pattern participants into single, domain-specific class. This has reduced the traceability of the patterns in the refined class diagram, a problem that has been identified by Soukup (1995) and Bosch (1998b) as a result of lack of patterns support in OO designs. But this may not be perceived as a major problem in constructing pattern-oriented designs and frameworks using POAD, because

- The framework is still documented using higher-level pattern diagrams that the designer can use in addition to the refined class diagram. Traceability between these diagrams and class diagrams could be supported by tools that automate the traceability process.

- When referring to the refined class diagram, the framework designer is interested in the design of the framework in terms collaborating classes. Knowing how patterns were used to build it may be of less interest at that level, especially when the designer is concerned about implementation of these classes. The higher-level design models in terms of pattern diagram are of more interest to the framework designer/maintainer because it is easier for the framework maintainer to work on a design with fewer design constructs (patterns) and more comprehensive documentation (pattern documentation templates) than to work on many design constructs (classes) whose documentation is not as rich.

Up to this point, POAD provided some design models that utilize UML capabilities to compose patterns. It does not provide all the models
that the designer needs to take this model into the implementation phase. To proceed with the development of the detailed class diagram, we can further analyze the refined class diagram using other UML models, such as statecharts and sequence diagrams. For instance, the dynamics of the refined class diagram can be studied using a UML collaboration diagram. Figure 11-13 illustrates a collaboration diagram to clarify the role of objects in the framework. Using this and other collaboration models, the designer can start defining message parameters, data types, additional methods, and so on.

**Figure 11-13. A collaboration diagram for the feedback design framework.**

As a result of analyzing the framework using interaction diagrams, collaboration diagrams, statecharts, and other UML models, we are able to add more design details to the refined class diagram of Figure 11-12. This could mean adding member functions for the framework classes, other properties or attributes to these classes, and references (pointers) in each class that implement the association relationships between that class and other classes in the framework. For example, from Figure 11-13 we added a MeasurePlant() method in the MeasurementSubject class and two methods in the ErrorObserver class, Analyze() and GetInput(). Figure 11-14 illustrates a more detailed class diagram for the framework.

**Figure 11-14. A detailed class diagram for the feedback control framework.**
Example: Quality Control in Production Lines

In this section, we illustrate a simple example for instantiating the framework in a quality-control system for a beverage bottle production line. In such systems it is usually required to distinguish defective bottles based on certain quality criteria defined by the system user. The system uses feedback data taken from actual measurements of the bottle under inspection to determine whether or not the criterion is met.

System Description

In the production line of the beverage bottle system, it is required to check the cleanliness of the bottles before filling them with the beverage. The objective is to pass the clean bottles that meet a certain criterion and reject dirty ones. Bottles are introduced into the system on a conveyer belt that transfers the bottle to the check unit at which the belt stops and measurements are taken. The measured data is processed to give a cleanliness value that is compared to a reference input and fed to the forward controller system. According to the error data values, the controller will either pass the bottle or remove it from the belt to be recycled. In either case the belt is activated again to proceed with processing the next bottle. From a physical point of view, the system consists of a conveyer belt, a bottle detector, a check unit containing a camera for taking images of the bottles, and actuators that remove dirty bottles from the belt. See Figure 11-15.

**Figure 11-15. An a feedback system for a beverage bottle system.**

Application-Specific Considerations
To simplify the example, we assume the following:

- The detector is a simple sensor that indicates the presence of the item.
- The implementation of the controller's routines will not be discussed. The required interface modules will depend on the operating system, the hardware, and the type of the sensor.
- The real-time behavior of the system is not discussed. Timing events may be required in other applications.
- Introducing items to the conveyor belt and removing them are out of the scope of the framework. We are more concerned with the feedback control aspects.

**Instantiation of the framework**

The following sequence of events takes place in the system:

- Bottles are fed to the conveyor belt, which stops as soon as a bottle is detected in the check area. Data collection is performed by recording an image of the bottle.
- Analysis of the image is conducted using specific image-processing routines. Measurements of cleanness and other attributes are calculated.
- The measured values are compared to reference inputs, and the error values are calculated.
- The forward controller rejects or passes the item according to the error data.

As discussed in Chapter 7, we can instantiate the framework in a specific application starting from the pattern-level diagram or the class-level diagram. Since this is an illustrative example and the framework user is the same as the framework designer (i.e., we have enough knowledge of the details of the framework), we instantiate the framework by starting with the framework class diagram in Figure 11-12. Following a simple approach for instantiating the framework, we start with each class in the framework class diagram and identify which elements need to be added as application-specific (the framework hooks) and which elements belong to the framework's fixed design.

- The **MeasurementSubject**. The function of this class is to measure the real-world data, which is the camera image taken of each bottle. The details of the image size and resolution are specified at the detailed design. The `MeasurePlant()` method is added to control the camera and record the image.
- The **MeasuredData**. It has the variables required to keep the instantaneous image of the bottle.
- The **FeedbackSubjectObserver**. This class handles the image taken of the bottle by invoking the `FBApply` method of the attached feedback control strategy `FBControlStrategyA`.
- The **FBControlStrategyA**. The necessary image-processing algorithm is implemented in this class. The `FBApply()` method implements an image-processing algorithm to detect the cleanness of the bottle and may also take some measurements of the image, such as width or height. The result of the algorithm is stored in a `FeedbackData` object.
- The **FeedbackData**. The necessary feedback data, such as the percentage of cleanness or the width, is represented in an object of this type.
- The **ErrorObserver**. After being notified by the `FeedbackSubjectObserver`, the object of this class gets the reference input data specified by the system user using the `GetInput()` method and compares it to the `FeedbackData`; the result is stored in an `ErrorData` object.
- The **Controller** class. The controller invokes the forward control strategy, which, depending on the cleanness percentage error, either removes the item from the belt via `RemoveItem()` if defective or passes the item via `PassItem()` if it is acceptable. The
Apply() method also invokes the StartBelt() procedure.

- When the belt conveys another bottle to the check unit, the detector initiates an interrupt to invoke the StopBelt() method of the MeasurementSubject.

By adding these application-specific methods to the class diagram model of Figure 11-12, we obtain Figure 11-16, which illustrates an instantiated class diagram for the application.

**Figure 11-16. The class diagram of the beverage bottle framework.**

From this example, we deduce the following:

- The instantiation process is direct and simple. The framework user does not add any new classes. She adds only application-specific methods and data types.
- Decreasing the dependencies between several parts of the framework adds simplicity to the design and increases the flexibility of using and instantiating the framework.
- The strategy for processing the image of the bottle is transparent to other design classes. This transparency is made possible by using the Strategy pattern.
Sample Implementation

This section contains a complete Java code implementation for the refined class diagram design of the feedback control framework. The implementation illustrates how the framework works for a hypothetical simple speed-control system. To simulate the system, we created a plant object that plays the role of speed sensor, which is controlled by a timer, and the timer triggers new data into the system. The input control data is adjusted by the user, and the speed converges to the requested speed. The complete model is shown in Figure 11-17.

```java
/************************************
* AbstractController.java
* 
*************************************/
public abstract class AbstractController
{
    protected Plant thePlant;
    public abstract void apply(DataHolder errorData);
    public abstract void FBApply(DataHolder inputData, DataHolder outputData);
    public AbstractController() {  }
    public void setPlant(Plant aPlant)  { thePlant = aPlant; }
}

/************************************
* ControlStrategyA.java
* 
*************************************/
public class ControlStrategyA extends AbstractController
{
    public ControlStrategyA() {  }
    public void apply(DataHolder errorData)
    {
        // This controller implements the forward strategy
        // Add your control strategy here
        // Get the error value
        int value  = errorData.getValue();
        // A simple control strategy
        if(value < 0)
        {
            // Apply control to the plant
            thePlant.control(true);
        }
        else if(value > 0)
        {
            // Apply control to the plant
            thePlant.control(false);
        }
        else // finally reached the requested speed
        {
            System.out.println("Plant is now regulated");
        }
    }
    public void FBApply(DataHolder inputData, DataHolder outputData)
    {
        // This controller only implements the feedforward strategy
    }
```
public class FBControlStrategyA extends AbstractController
{
    public FBControlStrategyA() {}
    public void apply(DataHolder errorData)
    {
        // This strategy does not provide a feedforward
        // control technique, only feedback control algorithms
    }
    public void FBApply(DataHolder inputData, DataHolder outputData)
    {
        // Your feedback control strategy goes here.
        // Takes the input data, processes it, and
        // returns an output data object
        // pass it as is for this prototype
        outputData.setValue(inputData.getValue());
    }
}

public class Controller
{
    protected AbstractController controlStrategy;
    protected Blackboard myBlackboard;
    protected DataHolder inputData;
    public Controller()
    {
        inputData = new DataHolder();
        inputData.setValue(65);
        // use 65 miles per hour just as an example
    }
    public void control(DataHolder feedbackData)
    {
        // Calculate the error
        DataHolder errorData = new ErrorData();
        errorData.setValue(feedbackData.getValue() - inputData.getValue());
        // Save it in the storage
        myBlackboard.setData(errorData);
        // Apply forward control strategy
        controlStrategy.apply(errorData);
    }
    public void setInputData(DataHolder newInputData)
    {
        inputData = newInputData;
    }
    public void setBlackboard(Blackboard theBlackboard)
    {
        myBlackboard = theBlackboard;
    }
    public void setStrategy(AbstractController aStrategy)
AbstractObserver.java

public interface AbstractObserver
{
    public void update(AbstractSubject changedSubject);
}

AbstractSubject.java

import java.util.Vector;

public abstract class AbstractSubject
{
    protected Vector observers;
    public void attach(AbstractObserver anObserver)
    {
        observers.addElement(anObserver);
    }
    public void detach(AbstractObserver anObserver)
    {
        observers.removeElement(anObserver);
    }
    public void notifyObservers()
    {
        int size = observers.size();
        for(int i = 0 ; i < size; i++)
        {
            ((AbstractObserver) observers.get(i)).update(this);
        }
    }
    public abstract DataHolder getState();
    public AbstractSubject()
    {
        observers = new Vector(2);
    }
}

ErrorObserver.java

public class ErrorObserver implements AbstractObserver
{
    private DataHolder errorData;
    protected Blackboard myBlackboard;
    public Controller theController;
    public ErrorObserver()
    {
        errorData = new ErrorData();
    }
    public void setBlackboard(Blackboard aBlackboard)
```java
myBlackboard = aBlackboard;

public void setController(Controller aController)
{
    theController = aController;
}

public void update(AbstractSubject changedSubject)
{
    errorData = changedSubject.getState();
    theController.control(errorData);
}

/*****************************
* FeedbackSubjectObserver.java
*
******************************/

public class FeedbackSubjectObserver extends AbstractSubject implements AbstractObserver
{
    protected AbstractController FBController;
    protected DataHolder feedbackData;
    protected Blackboard myBlackboard;

    public FeedbackSubjectObserver()
    {
        feedbackData = new DataHolder();
    }

    public void setFBController(AbstractController aController)
    {
        FBController = aController;
    }

    public void setBlackboard(Blackboard theBlackboard)
    {
        myBlackboard = theBlackboard;
    }

    public void update(AbstractSubject changedSubject)
    {
        // When notified, get the data from the
        // measurement subject
        // Role. Playing the role of a concrete observer class in
        // an Observer pattern
        DataHolder measuredData = changedSubject.getState();
        // Role. Playing the role of a context class
        // in a Strategy pattern
        FBController.FBApply(measuredData, feedbackData);
        // Role. Playing the role of a concrete subject class
        // in another Observer pattern
        myBlackboard.setData(feedbackData);
        notifyObservers();
    }

    public DataHolder getState()
    {
        return feedbackData;
    }
}

/*****************************
* MeasurementSubject.java
*
******************************/
```
public class MeasurementSubject extends AbstractSubject {
    private DataHolder measuredData;
    protected Plant thePlant;
    public MeasurementSubject()
    {
        measuredData = new MeasuredData();
    }
    public void measurePlant()
    {
        measuredData.setValue(thePlant.getPlantMeasurement());
        notifyObservers();
    }
    public void setPlant(Plant aPlant)
    {
        thePlant = aPlant;
    }
    public DataHolder getState()
    {
        return measuredData;
    }
}

public class Blackboard {
    protected DataHolder dataRecord[];
    public Blackboard() {
    }
    public void setData(DataHolder newData) {
    }
    public DataHolder getData() { return null; }
}

public class DataHolder {
    protected int currentValue;
    public DataHolder() {
    }
    public int getValue()
    {
        return currentValue;
    }
    public void setValue(int newValue)
    {
        currentValue = newValue;
    }
}

public class Plant {
    // Plant implementation
}

---

This code snippet represents a simple model for tracking measurements of a plant. It utilizes an observer pattern to update a subject whenever a measurement is taken. The `MeasurementSubject` class holds a `DataHolder` which is used to store the measurements. The `measurePlant` method sets the measurement and notifies observers. The `setPlant` method changes the associated plant. The `getState` method returns the current data holder.
import java.util.*;
public class Plant extends java.util.TimerTask {
    // initial value of speed
    protected int currentSpeed = 60;

    // start with 60 miles per hour
    protected MeasurementSubject aSensor;
    public Plant(MeasurementSubject theSubject) {
        aSensor = theSubject;
        // At the beginning, just assume the speed is
        // around one of the values
        currentSpeed += (Math.random() * 10 - 5);
    }

    public int getPlantMeasurement() {
        // randomize the speed value
        return currentSpeed;
    }

    public void control(boolean speedUp) {
        System.out.println("Current speed is: " + currentSpeed);
        System.out.println("Command received is: " + speedUp);
        if (speedUp) // speed up the car
            currentSpeed++;
        else {   // slow down
            currentSpeed--;
        }
    }

    public void run() {
        aSensor.measurePlant();
    }
}

import java.util.*;
public class aClient {
    protected java.util.Timer flushTimer;
    protected Plant aPlant;
    protected MeasurementSubject aMeasurementSubject;
    protected AbstractController forwardControl;
    protected AbstractController feedbackControl;
    protected Blackboard aBlackboard;
    protected ErrorObserver errorObserver;
    protected Controller aController;
    protected DataHolder inputData;
    protected FeedbackSubjectObserver feedbackSubjectObserver;
    public aClient() {
    }
    protected boolean isDone() {  return false;  }
    protected void run() {
        int returnResults = 0;
        // initialize the components
        returnResults = initialize();
    }
}
if (returnResults < 0)
{
    // Report error
    System.out.println("Cannot initialize the system");
    return;
}
// wait for user action to stop the system
while (!isDone())
{
    // sleep for a while and recheck again later
}
public static void main(String[] args)
{
    new aClient().run();
}
protected int initialize()
{
    aBlackboard = new Blackboard();
aMeasurementSubject = new MeasurementSubject();
aPlant = new Plant(aMeasurementSubject);
aMeasurementSubject.setPlant(aPlant);
forwardControl = new ControlStrategyA();
forwardControl.setPlant(aPlant);
feedbackSubjectObserver = new FeedbackSubjectObserver();
feedbackControl = new FBControlStrategyA();
feedbackSubjectObserver.setFBController(feedbackControl);
feedbackSubjectObserver.setBlackboard(aBlackboard);
aController = new Controller();
aController.setStrategy(forwardControl);
aController.setBlackboard(aBlackboard);
errorObserver = new ErrorObserver();
errorObserver.setBlackboard(aBlackboard);
errorObserver.setController(aController);

    // Attaching observers to subjects
    aMeasurementSubject.attach(feedbackSubjectObserver);
    feedbackSubjectObserver.attach(errorObserver);
    // Set the input control value
    inputData = new DataHolder();
    inputData.setValue(65);
aController.setInputData(inputData);
flushTimer = new java.util.Timer();
flushTimer.scheduleAtFixedRate(aPlant, 5000, 1000);
return 0;
}

Figure 11-17. A simple feedback control example.
null
Summary

In this chapter, we illustrated the applicability of the POAD process to develop design frameworks. Design frameworks developed using POAD are called pattern-oriented frameworks. The development process discussed in Chapter 7 is applied. A pattern-level is described as a higher design level of class diagram, which represents the framework in terms of patterns, components, and associations. Ordinary techniques represent the framework in terms of classes and their interaction. The approach of constructing a design framework using constructional design patterns as design components makes the framework easier to understand by providing different levels of abstraction and facilitating traceability between these levels.

In the following chapter, we discuss another example of applying POAD to develop a pattern-oriented design for simulating the behavior of waiting queues.
Chapter 12. Simulation of Waiting Queues

Simulation is a software engineering technique that is commonly used prior to actual implementation for several purposes, including verification of operations or assumptions, estimation of some statistical parameters about the application execution, checking the actual application operation for deadlocks or undesirable states, and studying the application behavior.

In this chapter we discuss the pattern-oriented development (using POAD) of a domain-specific architecture, which is used as a reference architecture for the development of applications that simulate the behavior of waiting queues. In these applications we deal with customers lining up for service from one or more service stations. Practical examples for such systems include the checkout counters at supermarkets, the airline check-in counters, immigration posts at the airport, and self-serve carwash stations. The purpose of developing such simulation environments is to be able to get some measurements about systems that implement waiting-queues structures. For instance, using the simulation environment, we are able to measure the quality of service, the fairness in treating customers, efficiency in service warranty, and productivity of the service stations.

We have developed an OO design for this architecture as part of an educational experiment in software reuse [Addy et al. 1999; Mili et al., 2001]. The architecture developed in Addy, Mili, and Yacoub's "A Case Study in Software Reuse" (1999) is not based on design patterns. In this chapter we describe how to use POAD to develop an equivalent pattern-oriented design for the reference architecture. We show how the domain architecture can be constructed using constructional design patterns as its building blocks. First, we explain the domain and application requirements, then we follow the POAD steps to create a pattern-oriented design for the waiting-queues simulator.
Background and Requirements

The focus of this case study is the domain of waiting-queue simulation. This is a more tightly scoped version of the general domain of discrete-event simulation. Other examples that fall under the general discrete-event simulation domain include communications systems such as telephone networks, where we deal with simulation of network nodes, distribution, packet delays, and other issues. Discrete-event systems also include traffic systems and manufacturing processes. For the purposes of this case study, we limit the discussion to servers, queues, and customers lining up for service.

Domain Engineering

The case study involves developing a domain-specific design for the purpose of producing applications that simulate the behavior of waiting queues. There are several degrees of variances that should be accommodated in the design:

- **Topology of service stations.** The number of service stations may be fixed or variable (in time). We may have one or more service station. If we have more than one service station, the service stations may be interchangeable (they deliver the same service) or not (e.g., checkout counters for 10 items or less, checkout counters for more than 10 items, etc.).

- **Service time.** The service time may be constant or variable. If it is variable, it may be determined by the customer, by the service station, or by both (e.g., customer determines amount of service needed, and service station prorates that with its own productivity factor). Also, if it is variable, it may be subject to a maximum service value (as is the case in CPU dispatching algorithms); when the maximum is reached, the customer may be dispatched out, queued at the end of the queue where it was, or considered a new arrival.

- **Topology of queues.** We may have a single queue, multiple interchangeable queues, or multiple queues with different service categories (each customer may line up at queues of a given category).

- **Typology of queues.** Queues could be first in, first out (FIFO) queues; last in, first out (LIFO) queues (stacks); priority queues; or limited size queues. Some applications (e.g., checkout counters) where customers may leave the waiting queue either in the order of their arrival (to receive service) or in reverse order (to change queues).

- **Arrival distribution.** Customers arrive to the system at various rates. The distribution of the customer arrivals in time could be Markovian distribution, Poisson distribution, or clustered distribution (if the service stations are immigration posts at an airport, then passenger arrivals are clustered around after flight arrivals).

- **Dispatching policy.** Customers are assigned to queues at random and may not change queues after the first assignment. Other policies might be that customers are assigned to the shortest queue upon arrival and may not subsequently change queues, or customers are assigned to the shortest queue upon arrival and subsequently switch queues to take the shortest until they are served.

- **Measurements.** As a result of the simulation, we are interested in obtaining one or more of the following measurements: the average waiting time (as a measure of quality of service); the standard deviation of waiting time (as a measure of fairness); the maximum waiting time (as a measure of service warranty); and the throughput (as a measure of productivity).

Application Engineering
In order to exercise the domain engineering activity that is carried out according to the requirements discussed above, the domain architecture, design, and components should be easily instantiable in a set of software applications that are all instances of the queue simulation domain. Examples of such applications include the following:

1. **CPU dispatching.** Simulation of the behavior of a CPU dispatching mechanism. In this application we measure fairness and throughput. There is a single priority queue with maximum service time (quantum service, Q); once a process has exhausted its service time, it is queued back with an increased priority.

2. **Self-serve car wash.** In this application we have a set of self-serve interchangeable car wash stations. Arriving cars line up at the shortest queue (queues of equal length are interchangeable) and do not subsequently change queues; queues are FIFO. Service is limited to a maximum value (but may take less time), and cars are expected to clear the station once the maximum time has expired. Arrival distribution could be, for instance, Markovian distribution. We are interested in monitoring maximum waiting time (we don't want anybody to leave before being served) and throughput (we want to serve as many people as possible).

3. **Checkout counters.** We have a number of checkout counters at a supermarket, some of which are reserved for shoppers with 10 items or less. Shoppers with 10 items or less line up in the shortest express checkout queue; others line up in the shortest queue reserved for them. Once they are lined up in some queue, shoppers do not leave the queue until it is their turn. Service time is determined by the shopper and by the productivity of the cash register attendant. Arrival rate is Markovian distribution. The number of stations increases whenever the longest queue exceeds a threshold value L and decreases by one whenever the number of stations of each category is greater than one and the length of the queue is zero. Whenever a new cash register is opened, shoppers at the end of the queue rush to line up at the station until the length of the queue equals the shortest current queue of the same type (express checkout, regular checkout). We are interested in average waiting time and fairness.

4. **Immigration posts.** We have a number of immigration stations at an airport, some of which are reserved for nationals; the others are for foreign citizens. There are two queues: one for nationals, the other for foreigners; each queue feeds into the corresponding set of stations, and there is no transfer between queues. The number of stations that handle nationals increases by one whenever the length of the nationals' queue exceeds some value, and the number of stations that handle foreigners increases by one whenever the length of the foreigners' queue exceeds some other value. The arrival rate is a clustered distribution, as passengers come by planeloads. Service time for nationals is constant, and service time for foreigners is determined by the passenger and by the productivity of the immigration agent attendant. We are interested in monitoring throughput.

5. **Check-in counters.** We have two FIFO queues for passengers at an airline check-in station: a queue for first class and a queue for coach. We have two categories of service stations: first class and coach; the number of stations does not change for the length of the experiment. The duration of the service is the same for all passengers and all stations of the same class, but differs from first class to coach. Passengers line up at their designated queue and do not leave it until they are served. Whenever one queue is empty, the corresponding service stations may serve passengers of the other queue (typically, first class stations serve coach passengers when no first-class passengers are awaiting). We are interested in monitoring the average waiting time and the maximum waiting time for each class of passengers.

Additional sets of applications that are considered in the same domain can be found in "A Case Study in Software Reuse" [Addy et al., 1999].
POAD Analysis for the Waiting-Queues Simulation Architecture

Requirements Analysis

Based on the above requirements, we find that the architecture we are developing is specific for simulation. Simulation architectures are often centered around a dynamic event list as the communication vehicle between cooperating components. To reduce interdependency between components, an event-trigger architecture is usually selected. An essential activity of an event-driven environment is identifying possible events, their initiator, and their processing destinations. The simulation advances event by event on a time axis and gathers relevant statistics. Event is one key abstraction in the simulation architecture. The domain analysis identifies a set of events that depict all actions necessary for the execution of the simulation along with the components that generate the event and the components that take action accordingly. Therefore, an event management component is needed to orchestrate the execution and communication of events.

In addition to the event management conceptual component, we can identify several other components from the requirements. For example, any of these simulation applications is used to simulate waiting queues; therefore, the concept of one or more waiting queues should be captured in a conceptual component by itself.

In these simulation applications, customers and their attributes are generated according to some specific arrival distributions. The generation of customers and their attributes could also be captured in another component.

Customers generated by a customer generator component and waiting in one of the queues will eventually be served by some service unit. The attributes and number of the service units will vary from one simulation application to another; however, the concept of a service unit should be captured by a conceptual component by itself.

In the following we summarize the conceptual components that we identify and their responsibilities. Since the size of the application seems manageable at the requirement analysis phase, we will not use any grouping mechanisms (such as subsystems or packages).

- **Customer generator.** The customer generator component uses one of the distribution functions designated in the domain description to generate the time of arrival of the next customer. Appropriate distributions are used to generate other characteristics of the customer, such as the number of service units (or service time) needed, the type of the customer, and information on the service received. The generation of customer is placed as an event and is introduced to the event management component. The distribution function that determines the time of the next customer arrival varies according to applications, as well as the generation of application-specific attributes for the customer such as the service time needed and the category of service.

- **Queue facility.** The queue facility component consists of a set of queue categories, where each queue category contains one or more queues. Queue categories are used to capture the different types of queues, such as express or normal queues. The queues are identifiable such that the servers can identify the queue that holds customers for them. Any event that indicates an action for a queue category or a queue is delegated to the queue facility, which passes the event to the appropriate queue category. The queue category delegates events to the appropriate queue.

- **Service facility.** The service facility component consists of a set of service categories, where each service category contains one or more servers. Service categories are used to capture the different types of servers such as business or economy classes of service. The servers are identifiable and should be attached to one or more queues that they serve. Any event that indicates an action for a service category or a server is delegated to the service facility and in turn passes the event to the server category and to the specified server as appropriate.

- **Event manager.** The event manager component serves as the main driver for the simulation. We refer to it as the simulation driver or scheduler. It repeatedly processes events and delegates the actions to one or more components. The schedule manager is the most domain-specific of all the reusable components. At the end of the simulation, the schedule manager
calculates the metrics for the simulation run and prints a final report. It records the information contained within a customer each time a customer completes its number of service units in a server. It calculates the averages and totals specified in the domain engineering description.

As a result of the requirements analysis activity, we have identified the need for the following conceptual components: a service facility, a queuing facility, a customer generator, and an event manager.

**Acquaintance and Retrieval**

In POAD, the analyst should consider off-the-shelf design pattern libraries. The analyst browses existing pattern databases for solution fragments (patterns) to use in developing the overall application design. The simulation application domain is a reactive system in which components react to event and generate other events. Therefore, the analyst may want to get acquainted with design patterns for reactive systems, such as those in the works of Schmidt and colleagues (1995b, 1998, 2000). For a list of design patterns that fall under the reactive and real-time system categories, refer to Linda Rising's *The Pattern Almanac* (2000, pp. xxxix).

Since the simulation architecture is heavily based on queues, we might also want to consider the patterns for designing interactive applications using queues [Wake et al. 1996]. The Event Queue pattern in this pattern collection illustrates common ways of designing the event object to be used in the simulation.

In addition to patterns for reactive systems, we also consider general-purpose design patterns, since some of the problems we are solving are general design problems. For instance, consider the service facility component, which is composed of service categories, which in turn are also composed of server units. This composition and hierarchy could be addressed by general-purpose design patterns that address composition and hierarchy [e.g., Gamma et al. 1995; Buschmann et al. 1996].

**Pattern Selection**

Based on the responsibilities and functionalities assigned to each conceptual component, we identify and select patterns that could provide solutions to implement these responsibilities.

The Queue Facility component is composed of a set of queue categories, and each queue category is composed of one or more queues. This composite and hierarchical structure is common to many software designs; therefore, we search for a general-purpose design pattern that implements a hierarchical and composite design. We select the Composite pattern [Gamma et al. 1995, pp. 163] to be used in implementing the composite design of the queue facility. The Composite pattern "composes objects into tree structures to represent part-whole hierarchies."

The Service Facility component is composed of a set of service categories, and each service category is composed of one or more servers. Similar to the design of the queue facility, this server composition and hierarchical structure is common to many software designs. We select another Composite pattern to be used in implementing the composite design of the service facility.

In the Customer Generator component, we use a distribution function to generate the next customer to arrive together with the simulation attributes for that customer, such as its type and its service request. Since this common architecture will be instantiated in many waiting-queues simulation applications, the generation of customer should be kept flexible to allow applications to hook in specific generation strategies. There are several patterns that can be used to implement this flexible strategy for customer generation; for instance, we can use the Strategy pattern or the TemplateMethod pattern. For this case study, we select the TemplateMethod pattern [Gamma et al. 1995, pp. 325] as an interface for generating customers. The TemplateMethod pattern defines the skeleton of an algorithm (in our case study the steps required to generate a customer and the associated parameters) and defers the implementation of each step of the algorithms to subclasses without changing the algorithm structure as sequence of steps.

The Event Manager (also called Scheduler) component plays the role of demultiplexing and dispatching of events. It receives events, identifies the appropriate event handler, and dispatches the event to the selected handler. Handling events is a domain-specific function that is related to reactive, distributed, and event-driven systems. In searching for domain-specific libraries of patterns for reactive and event-driven systems, we select the Reactor pattern to implement the design of the Event Manager. The Reactor pattern supports the "de-multiplexing and dispatching of events to multiple event handlers triggered concurrently by multiple events, and simplify event-driven
applications by integrating the de-multiplexing of events and the dispatching of the corresponding event handler” [Schmidt 1995b].

In the analysis phase, we have selected a set of patterns that fulfils the responsibilities identified for each conceptual component. In choosing these patterns, we considered how the pattern solves the design problem. In summary, a Composite pattern is selected for the Service Facility component, another Composite pattern is selected for the Queue Facility, a TemplateMethod pattern is selected for the generation of customers, and the Reactor pattern is selected to implement the Event Manager.
In this step, we create instances of the selected patterns and identify the relationships between these instances. As a result, a pattern-level logical view of the system is developed.

First, we create pattern instances. In the analysis phase, we selected the Composite pattern to design the queue facility. Therefore, we use the pattern instance name QueuingFacility of type Composite to implement the queue facility. Similarly, we selected the Composite pattern to design the service facility. Therefore, we use the pattern instance name ServiceFacility of type Composite to implement the service facility. We have also selected the TemplateMethod pattern to implement the design for the customer generator. We use the pattern instance name Generator of type TemplateMethod pattern to implement customer generation. We also selected the Reactor pattern to implement the design of the event management components. Therefore, we use the pattern instance name Scheduler of type Reactor pattern to implement the event list and the events scheduling and dispatching. As a result, the Pattern-Level diagram will contain four pattern instances: QueuingFacility, ServiceFacility of type Composite; Generator of type TemplateMethod; and Scheduler of type Reactor.

Second, we define the dependency relationship between the pattern instances. The Scheduler dispatches events to the ServiceFacility and the QueuingFacility to handle relevant events accordingly. In turn, ServiceFacility and QueuingFacility use the Scheduler to schedule new events generated in the course of their processing. The Scheduler uses the Generator to generate new customer arrivals. Finally, we use the pattern instances and their relationships to construct the Pattern-Level diagram, as shown in Figure 12-1.

Figure 12-1. A Pattern-Level diagram for the waiting-queues simulator architecture.
be used in Figure 12-1 to represent the EventQueue pattern together with the possible pattern dependencies with the Scheduler pattern instance or other patterns.

Constructing Pattern-Level with Interfaces Diagrams

In this step, we analyze the relationships between pattern instances. The dependency relationship between patterns in the pattern-level view is a conceptual, high-level dependency relationship. We further trace these dependencies to lower-level design.

First, we declare interfaces for the patterns used in the Pattern-Level diagram. The interface of a Composite pattern is the Component class by which its constituting elements, whether simple (leaf child) or composite (parent), interface to other components. Therefore, each of the two pattern instances ServiceFacility and QueuingFacility will have the interface Component. The Reactor pattern has two types of interfaces: the EventHandler interface class for delegating the processing of events to other classes and the Dispatch() interface operation for receiving and scheduling events. The interface of a TemplateMethod pattern is TemplateMethod() of the class AbstractClass, which contains the skeleton of the customer-generation steps implemented by concrete subclasses according to various customer-generation policies.

Then, we identify the relationship between pattern interfaces by translating all dependency relationships between patterns in a Pattern-Level diagram of Figure 12-1 into relationships between interface classes and/or interfaces operations. The Component interface class of the ServiceFacility and the QueuingFacility pattern instances has a relationship with the Dispatch() interface operation of the Scheduler pattern instance to schedule the processing of events. The EventHandler interface class of the Scheduler pattern instance has a dependency relationship with the Component interface classes of the ServiceFacility and the QueuingFacility pattern instances to handle the processing of events. The Dispatch() interface operation of the Scheduler pattern instance has a dependency relationship with the TemplateMethod() interface operation of the Generator pattern instance. The product of this process is the Pattern-Level with Interfaces diagram. Figure 12-2 illustrates the Pattern-Level with Interface diagram for the simulation architecture.

Figure 12-2. A Pattern-Level with Interfaces diagram for the waiting-queues simulator.

Constructing Detailed Pattern-Level Diagrams

To construct the Detailed Pattern-Level diagram, we express the internal parts of each instantiated pattern from the Pattern-Level with Interfaces diagram. Since we use pervasive design patterns in developing the Pattern-Level diagram, their structure can be found in the literature [Gamma et al. 1995, Schmidt 1995b]. Figure 12-3 illustrates the Detailed Pattern-Level diagram.
Figure 12-3. A Detailed Pattern-Level diagram for waiting-queues simulator.

Note that we do not make any additional design decisions in this step. With the appropriate tool support, Figure 12-3 is a direct generation from the Pattern-Level with Interfaces diagram by simply retrieving the internal class diagram model for each pattern from a pattern database.
POAD Design Refinement for the Waiting-Queues Simulation Architecture

Instantiating Pattern Internals

In this step, we add application-specific nature to the Detailed Pattern-Level diagram by renaming internal pattern classes according to the application design environment, choosing names for pattern participants that are meaningful in the application context, and defining application-specific names for operations in patterns.

Instantiating the internals of the ServiceFacility is illustrated in Figure 12-4. The ServiceFacility pattern instance is composed of

- **ServiceComponent.** A ServiceComponent is an abstract interface for all service elements. A service element can be a Server, a ServiceCategory, or the whole ServiceFacility. It defines the method signatures for serving a customer (Serve()) and completion of the service (ServiceComplete()), which are the Operation() methods in the abstract definition of the Composite pattern. The Add() and Remove() operations in the abstract Composite pattern definitions are now renamed into application-specific methods: the AddServer() and RemoveServer() methods. Methods are implemented in concrete subclasses according to the type of the service component.

- **Server.** The actual service station that serves a customer. The Server class implements the methods of the abstract class ServiceElement. It contains attributes related to the productivity of the server, its status, and maximum service time. The Server class is the leaf node that does not contain any other service elements.

- **ServiceCategory.** A ServiceCategory class is a composition of Server classes that are of the same category. The type of the server category depends on the application implementing the simulation architecture. The ServiceCategory contains attributes defining the category and the number of servers. It implements the methods related to adding a server to its pool of servers.

- **ServiceFacility.** The manager of the whole service station and categories. It is composed of several service categories, each of which is composed of a number of servers. It has attributes defining the number and type of service categories that the facility contains. In each application there will usually be one ServiceFacility that contains the rest of the service elements. It implements the interfaces to add other service components, such as other servers or other service categories.

**Figure 12-4. Instantiating the ServiceFacility pattern.**
Instantiating the internals of the QueuingFacility is illustrated in Figure 12-5. The QueuingFacility pattern instance is composed of

- **QueuingComponent.** An abstract interface for all queuing elements. A QueuingElement can be a Queue, a QueueCategory, or the whole QueueFacility. It defines the interface for queuing Enqueue() and dequeuing Dequeue() a customer, which are the Operation() methods in the abstract definition of the Composite pattern. The Add() and Remove() operations in the abstract Composite pattern definitions are now renamed into application-specific methods: the AddQueuingElem() and RemoveQueuingElem() methods. Methods are implemented in concrete subclasses according to the type of the queuing component.

- **Queue.** The actual queue that holds a customer. The Queue class implements the methods of the abstract class QueuingElement. It contains attributes related to the size of queue and its type. The Queue class can be further subclassed to implement different types of queues such as FIFO, LIFO, or priority queues.

- **QueueCategory.** A composition of queues that are of the same category. It contains attributes defining the category and number queues in this category. The type of the queue category depends on the application implementing the simulation architecture. The QueueCategory class implements the methods related to adding a queue to its pool of queues.

- **QueueFacility.** The manager of the whole queuing system. It is composed of several queue categories, each of which is composed of number of queues. It has attributes defining the number and type of queue categories that the facility contains. In each application there will usually be one QueueFacility that contains the rest of the queuing elements. It implements the interfaces to add other queue components such as other queues or other queue categories.

Figure 12-5. Instantiating the QueuingFacility pattern.

Instantiating the internals of the Scheduler is illustrated in Figure 12-6. The Scheduler pattern instance is composed of

- **EventList.** The Reactor class in the Reactor pattern is renamed to EventList. The EventList is the main component of the scheduler that holds a repository of all events that occur in the system. Events in the EventList are time ordered. The EventList supports scheduling of new events and dispatching events to be processed to the appropriate event handler. There are several other attributes and methods of the EventList that are not shown in the diagram for simplicity. Moreover, we are only interested at this level in design elements that are crucial for integration with elements of other patterns.

- **EventHandler.** An EventHandler is an abstract interface to all possible concrete event handlers that can process and handle events. It defines interfaces to query the status of the handler, initiate the handling of events, or terminate the handling of
ConcreteEventHandler. A ConcreteEventHandler class is the concrete implementation of the EventHandler interface. In this waiting-queues simulation architecture, the concrete event handlers are those handlers that handle simulation events, which are the two interfaces QueuingComponent and ServiceComponent.

**Figure 12-6. Instantiating the Scheduler pattern.**

Instantiating the internals of the Generator is illustrated in Figure 12-7. The Generator pattern instance is composed of

- **AbstractGenerator.** AbstractGenerator is the interface class that defines the steps for generating a customer arrival event. The TemplateMethod() in the abstract TemplateMethod pattern is renamed to GenerateArrival(). The GenerateArrival() method is the template method that defines the procedures for generating a customer-arrival event object. GenerateArrival() method delegates the generation of specific customer attributes to concrete classes by calling abstract methods, which are implemented in subclasses such as ArrivalTimePolicy(), ArrivalCategoryPolicy(), and ServiceUnitsPolicy().

- **RandomArrival and MarkovianArrivals.** These are concrete implementations of the arrival policies. They provide several concrete implementations of the arrival generator based on an arrival policy and variations in the techniques used to generate the customer attributes.

**Figure 12-7. Instantiating the Generator pattern.**
Developing an Initial Class Diagram

Starting from the Detailed Pattern-Level diagram in Figure 12-3, we use pattern interfaces and the instantiated details of pattern internals to construct a UML class diagram. This is a simple process of combining the domain-specific details with the Detailed Pattern-Level diagram. The class diagram developed at this phase is an initial step to develop the static design model of the pattern-oriented architecture for the waiting queues simulation applications. In this step we convert relationships between pattern interfaces in the Detailed Pattern-Level to relationships between internal classes of instantiated patterns.

First, we convert the class/class relationship between pattern interfaces by tracing each of the two class interfaces to the internal classes of the pattern instances. For example, Figure 12-8(a) is taken from Figure 12-3 to illustrate the class/class relationship between the EventHandler interface class of the Scheduler pattern instance of type Reactor and the Component interface class of the ServiceFacility pattern instance of type Composite. Figure 12-8(b) illustrates the same relationship after instantiating the internal details of each of the two patterns which has now become a relationship between the two interface classes EventHandler and ServiceComponent (after renaming the internal details of the two patterns). We then translate this relationship into UML class association relationships between internal classes of both patterns. The EventHandler interface is traced inside the instantiated Scheduler to the internal class EventHandler, and the ServiceComponent interface class is traced inside the instantiated ServiceFacility to the ServiceComponent class. Hence, a UML class association relationship is established between the class EventHandler and the class ServiceComponent, as shown in Figure 12-8(c).

Figure 12-8. Converting the class/class pattern interface relationship into UML class association.
Similarly, we convert the class/class relationship between the `EventHandler` interface class of the Scheduler pattern instance of type Reactor and the `Component` interface class of the QueuingFacility pattern instance of type Composite. As a result, a UML class association relationship is established between the class `EventHandler` of the Scheduler pattern instance and the class `QueuingComponent` of the Queuing Facility pattern instance.

Second, we convert the operation/class relationship between pattern interfaces by tracing each interface operation to the internal pattern class that implements the operation. The interface class is also traced to the pattern’s internal class, and hence a class relationship is established between internal classes of the two interfacing patterns. For example, Figure 12-9(a) is taken from Figure 12-3 to illustrate the operation/class relationship between the `Dispatch()` interface operation of the Scheduler pattern instance of type Reactor and the `Component` interface class of the ServiceFacility pattern instance of type Composite. Figure 12-9(b) illustrates the same relationship after instantiating the internal details of each of the two patterns, which has now become a relationship between the interface operation `Dispatch()` and the interface class `ServiceComponent` (after renaming the internal details of the two patterns). We then translate this relationship into UML class association relationships between internal classes of both patterns. The `Dispatch()` interface operation is traced inside the instantiated Scheduler to the internal class `EventList`, and the `ServiceComponent` interface class is traced inside the instantiated ServiceFacility to the `ServiceComponent` class. Hence, a UML class association relationship is established between the class `ServiceComponent` and the class `EventList`, as shown in Figure 12-9(c).

**Figure 12-9. Converting the operation/class pattern interface relationship into UML class association.**
Third, we convert the operation/operation relationship by tracing each interface operation to the internal class of the pattern. These internal classes of the two interfacing patterns will have a class association relationship. For example, Figure 12-10(a) is taken from Figure 12-3 to illustrate the operation/operation relationship between the Dispatch() interface operation of the Scheduler pattern instance of type Reactor and the TemplateMethod() interface operation of the Generator pattern instance of type TemplateMethod. Figure 12-10(b) illustrates the same relationship after instantiating the internal details of each of the two patterns, as discussed in section 12.4.1, which has now become a relationship between the interface operation Dispatch() and the interface operation GenerateArrival() (after renaming the internal details of the two patterns). We then translate this relationship into UML class association relationships between internal classes of both patterns. The Dispatch() interface operation is traced inside the instantiated Scheduler to the internal class EventList, and the GenerateArrival() interface operation is traced inside the instantiated Generator to the AbstractGenerator class. Hence, a UML class association relationship is established between the class AbstractGenerator and the class EventList, as shown in Figure 12-10(c).

Figure 12-10. Converting the operation/operation pattern interface relationship into UML class association.
As a result, the design is now viewed as a UML class diagram, as shown in Figure 12-11. The class diagram that is produced is an initial step to develop the static design model of the simulator.

Figure 12-11. The initial class diagram for the waiting queues simulator.
It can be easily recognized that the patterns are still notable in the class diagram as shown by the dotted boxes around the classes. We recall that we do not discard earlier diagrams. As part of POAD, all the models in figures 12-1 through 12-11 are saved as analysis and design models. It is the role of a tool support to save these models and provide the necessary traceability mechanisms between them.

How About Code Generation?

In the above process, we do not generate code at each level and keep it in synchronization with other levels. The process is mostly about manipulating the design elements (patterns, classes, methods, etc.) at that analysis and design level. Code generation comes later, after the refined design is complete. It is also unnecessary to generate code from the initial class diagram, because in the following phases the designer will change the design of the system by merging and grouping several classes together. So, if you are eager to see code samples, wait until you have completed the following phase.

Design Optimization

The class diagram obtained from gluing patterns together at the high-level design may not be dense or profound because we just strung the patterns together. It could possibly have replicated abstract classes due to the fact that we use multiple instances of the same pattern type. For example, we used the ServiceFacility and the QueuingFacility instances of type Composite pattern. The design could also have some classes with trivial responsibilities because these classes are there in the design to forward messages to internal participants of the pattern. Therefore, in this step we use reduction and grouping mechanisms to optimize the UML design diagrams obtained in the previous step.

The design obtained in Figure 12-11 can be further refined using reduction or grouping. Let’s consider first the possibility of reduction by removing replicated abstract classes. The Composite pattern is instantiated twice, once for the ServiceFacility and again for the QueuingFacility. The Composite pattern has an abstract class Component, which could have two occurrences because we use two instances of the same pattern type Composite. However, for this application, the two Component classes—ServiceComponent and QueuingComponent—of the two pattern instances cannot be merged together because the interfaces offered by the two abstract classes differ from being a service component to a queuing component; hence, we cannot merge these two abstract classes into one class.

Next, we consider the possibility of optimizing the design by merging classes with trivial responsibilities. The EventHandler class of the Scheduler pattern instance is the interface for the concrete event handlers that handle specific events. In this application, the event handlers are either the QueuingFacility pattern instance or the ServiceFacility pattern instance. The two abstract classes ServiceComponent and QueuingComponent can play the role of event handlers, and hence the EventHandler class and the ConcreteEventHandlers can be replaced by the two classes ServiceComponent and QueuingComponent. Events are dispatched from the EventList to either of the two classes such that the HandleEvent() method is an actual domain-specific method, such as Serve() or ServiceComplete() methods in the ServiceComponent class and Enqueue() or Dequeue() methods in the QueuingComponent class. As another design alternative, the designer might want to keep the EventHandler interface explicit; hence, the EventHandler class becomes the interface class that the two classes ServiceComponent and QueuingComponent implement. In the following design diagrams, we use the first design (replace the EventHandler classes with the ServiceComponent and QueuingComponent classes) to reduce the inheritance hierarchy.

We can further refine the design by using a SimulatorDriver class that continuously retrieves events from the EventList and delegates its processing to the appropriate component. In this case, the EventList is considered a passive, time-ordered list. It provides two methods: getNextEvent() to get the top element of the timely ordered list and Dispatch(), which is used by a ServiceElement or a QueuingElement to schedule an event in the list. The refined class diagram is shown in Figure 12-12.
As mentioned in Chapter 9, it could become difficult to identify the patterns at this level because the design is now represented in terms of domain-specific classes. This problem has always existed in many techniques that use patterns directly at the class diagram level without developing higher-level design models. POAD has one particular advantage. When applying POAD, the designer keeps all the models developed throughout the development lifecycle. These models are traceable bottom-up from the class level (Figure 12-12) to the pattern level (Figure 12-1) and top-down from the pattern level to the class level. With the appropriate tools, this traceability can be automated in a design environment.

As an example of top-down traceability, the Reactor pattern in Figure 12-1 is traced to the classes EventList (as a Reactor), ServiceComponent (as an Event-Handler), and QueuingComponent (as an EventHandler) in Figure 12-12.

As an example of a bottom-up traceability, the QueuingComponent in Figure 12-12 is traced up to be the Component of the QueuingFacility pattern instance and the EventHandler of the Scheduler pattern instance in Figure 12-1.
Sample Implementation

The final class diagram from Figure 12-12 can be instantiated in many applications that simulate the behavior of waiting queues. This section gives the reader the complete model and sample code in Java for an application that utilizes the above framework in simulating the behavior of check-in waiting queues in airports. We will use one queue and one server for business-class service and one queue and one server for coach-class service. The instantiated model for the application is illustrated in Figure 12-13 and the Java implementation follows.

```java
package WaitingQueue;

public class SimulatorDriver
{
    protected EventList eventlist;
    protected AbstractGenerator customerGen;
    protected Measurement TheMeasurement;
    public SimulatorDriver()
    {
        eventlist = EventList.getInstance();
        TheMeasurement = Measurement.getInstance();
        customerGen = new RandomArrivals();
    }

    public void runSimulation()
    {
        // Create the application by:
        // 1) Instantiating the domain components
        // 2) Establishing the relationship between components
        // Create basic components: ServiceFacility, QueueFacility
        QueueFacility q_facility = new QueueFacility(2 /*Maximum Number of Queue Categories in the Facility*/);
        ServiceFacility s_facility = new ServiceFacility(2 /*Maximum Number of Service Categories in the Facility*/);
        // Create Service Categories and add them to the service facility
        s_facility.addServiceCategory(1 /*MaximumNum of Servers in this Category*/, AllowedCategoriesEnum.COACH);
        s_facility.addServiceCategory(1 /*MaximumNum of Servers in this Category*/, AllowedCategoriesEnum.BUSINESS);
        // Create Queue Categories and add them to the queuing facility
        q_facility.addQueueCategory(AllowedCategoriesEnum.COACH, false /*Non REORDABLE*/);
        q_facility.addQueueCategory(AllowedCategoriesEnum.BUSINESS, false /*Non REORDABLE*/);
        // Add queues to the queuing facility; one queue for COAH and one for BUSINESS
        int TheQueueID_1 = q_facility.addQueue (AllowedCategoriesEnum.COACH);
        int TheQueueID_2 = q_facility.addQueue (AllowedCategoriesEnum.BUSINESS);
```
// Add servers to the service facility;
// one server for COACH and one for BUSINESS
s_facility.addServer(AllowedCategoriesEnum.COACH, 1 /* Productivity*/, 30 /*Maximum Service Time*/, TheQueueID_1);
s_facility.addServer(AllowedCategoriesEnum.BUSINESS, 1 /* Productivity*/, 30 /*Maximum Service Time*/, TheQueueID_2);

// Initialization:
Event firstevent = new Event();
firstevent.Type = EventTypeEnum.ARRIVAL;
firstevent.EventTime = 0; // Initially at zero time
firstevent.Category = AllowedCategoriesEnum.COACH;
Customer aCustomer = new Customer(0 /*ArrivalTime*/, 5 /*ServiceUnits*/,
AllowedCategoriesEnum.COACH/*Category*/);
firstevent.CarriedCustomer = aCustomer;
eventlist.Dispatch(firstevent);
// Initialize a simulation period
long TotalSimulationTime = 100;

// Run the simulation
// 1. Dispatch an event from the eventlist
// 2. Increment the simulation time to the event time.
// 3. Call the simulate function of the node specified in
// the event body.
// 4. Repeat from step 1 unless the simulation time is over.
Event sim_event;
while(true)
{
    sim_event = eventlist.GetNextEvent();
    if(sim_event.EventTime > TotalSimulationTime )
        break;

    // Simulate the event according to the event type.
    if(sim_event.Type == EventTypeEnum.ARRIVAL)
    {   //Generate another ARRIVAL
        aCustomer = new Customer(0,0,AllowedCategoriesEnum.COACH);
customerGen.generateArrival(sim_event.EventTime, aCustomer);
q_facility.Enqueue(sim_event.CarriedCustomer);
    }
    if(sim_event.Type == EventTypeEnum.SERVE)
    {
        s_facility.serve(sim_event.CarriedCustomer, sim_event.EventTime, sim_event.
ServerID, sim_event.Category);
    }
    if(sim_event.Type == EventTypeEnum.DEQUEUE)
    {
        q_facility.Dequeue (sim_event.Category, sim_event.FromQueueID, sim_event.
EventTime,sim_event.ServerID);
    }
    if(sim_event.Type == EventTypeEnum.SERVICE_COMPLETE)
    {
        s_facility.serviceComplete(sim_event.EventTime, sim_event.ServerID,sim_event.
Category);
    }
    if(sim_event.Type == EventTypeEnum.CHECK_SERVER)
    {
        s_facility.checkServer(sim_event.EventTime, sim_event.FromQueueID, sim_event.
Category);
    }
    if(sim_event.Type == EventTypeEnum.REORDER)
    {
        q_facility.ReorderCategory(sim_event.Category);
    }
}
// Print Statistics
System.out.println("Number of Customers Served : " + TheMeasurement.NumCustomers + "
\n");
System.out.println("Average Waiting Time : " + TheMeasurement.AverageWaitingTime() + "\n");
System.out.println("Maximum Waiting Time: " + TheMeasurement.MaxWaitingTime() + "\n");
System.out.println("Standard Deviation of Waiting Time : " + TheMeasurement.StdWaitingTime() + "\n");
System.out.println("Throughput : " + (float)((float)TheMeasurement.NumCustomers/(float)TotalSimulationTime) + "\n");

public static void main(String[] argc)
{
    SimulatorDriver aSimulator = new SimulatorDriver();
aSimulator.runSimulation();
}

/*********************************************************
* ServiceComponent.java
* **********************************************************/
package WaitingQueue;
public class ServiceComponent
{
    public void addServiceCategory(int maxNumberOfServers, AllowedCategoriesEnum aCategory){}
    public void removeServiceCategory(AllowedCategoriesEnum aCategory) {}
    public void addServer(AllowedCategoriesEnum aCategory, float productivity, long maxServiceTime, int queueID) {}
    public void removeServer(AllowedCategoriesEnum aCategory, int queueID) {}
    public void serve(Customer aCustomer, long currentTime, int serverID, AllowedCategoriesEnum aCategory) {}
    public void serviceComplete(long currentTime, int serverID, AllowedCategoriesEnum aCategory) {}
    public void checkServer(long currentTime, int fromQueueID, AllowedCategoriesEnum aCategory) {}
    public ServiceComponent() {}
}

/**********************************************************
* ServiceFacility.java
* **********************************************************/
package WaitingQueue;
public class ServiceFacility extends ServiceComponent
{
    protected ServiceComponent serviceCategories[];
    protected int MaxNumCategories;
    protected int NumCategories;
    public ServiceFacility(int _MaxNumCategories)
    {
MaxNumCategories = _MaxNumCategories;
serviceCategories = new ServiceCategory[MaxNumCategories];
NumCategories = 0;
}

public void addServiceCategory(int maxNumberOfServers, AllowedCategoriesEnum aCategory)
{
    if(NumCategories < MaxNumCategories)
    {
        serviceCategories[NumCategories] = new ServiceCategory(maxNumberOfServers,
                                   aCategory);
        NumCategories++;
    }
}

public void addServer(AllowedCategoriesEnum aCategory, float productivity, long maxServiceTime, int queueID)
{
    int i=0;
    for(i=0;i<NumCategories;i++)
        if(((ServiceCategory) serviceCategories[i]).GetCategory() == aCategory)
            ((ServiceCategory) serviceCategories[i]).addServer(aCategory, productivity,
                                          maxServiceTime, queueID);
        break;
    }
}

public void removeServer(AllowedCategoriesEnum aCategory, int queueID) {}

public void serve(Customer aCustomer, long currentTime, int serverID, AllowedCategoriesEnum aCategory)
{
    int i=0;
    for(i=0;i<NumCategories;i++)
        if(((ServiceCategory) serviceCategories[i]).GetCategory() == aCategory)
            ((ServiceCategory) serviceCategories[i]).serve(aCustomer, currentTime,
                                          serverID, aCategory);
        break;
    }
}

public void serviceComplete(long currentTime, int serverID, AllowedCategoriesEnum aCategory)
{
    int i=0;
    for(i=0;i<NumCategories;i++)
        if(((ServiceCategory) serviceCategories[i]).GetCategory() == aCategory)
            ((ServiceCategory) serviceCategories[i]).serviceComplete (currentTime,
                                          serverID, aCategory);
        break;
    }
}

public void checkServer(long currentTime, int fromQueueID, AllowedCategoriesEnum aCategory)
{
    int i=0;
    for(i=0;i<NumCategories;i++)
        if(((ServiceCategory) serviceCategories[i]).GetCategory() == aCategory)
            ((ServiceCategory) serviceCategories[i]).checkServer (currentTime,
                                          fromQueueID,aCategory);
        break;
    }
package WaitingQueue;

public class ServiceCategory extends ServiceComponent {
    protected EventList eventlist;
    protected ServiceComponent servers[];
    protected int MaxNumServers;
    protected int NumServers;
    protected int ServerIDs;
    protected AllowedCategoriesEnum Category;

    public ServiceCategory(int _MaxNumServers, AllowedCategoriesEnum _Category) {
        Category = _Category;
        MaxNumServers = _MaxNumServers;
        servers = new ServiceComponent[MaxNumServers];
        NumServers = 0;
        ServerIDs = 0;
        eventlist = EventList.getInstance();
    }

    public void addServer(AllowedCategoriesEnum aCategory, float productivity, long maxServiceTime, int queueID) {
        if(NumServers!=MaxNumServers) {
            servers[NumServers] = new Server(productivity, maxServiceTime, queueID);
            ((Server) servers[NumServers]).ServerID = ServerIDs;
            ((Server) servers[NumServers]).myCategory = Category;
            ServerIDs++; // To guarantee uniqueness
            NumServers++;
        }
    }

    public void removeServer(AllowedCategoriesEnum aCategory, int queueID) {
    }

    public void serve(Customer aCustomer, long currentTime, int serverID, AllowedCategoriesEnum aCategory) {
        int i=0;
        for(i=0;i<NumServers;i++)
            if(((Server) servers[i]).ServerID == serverID)
                ((Server) servers[i]).serve(aCustomer, currentTime, serverID, aCategory);
        }

    public void serviceComplete(long currentTime, int serverID, AllowedCategoriesEnum aCategory) {
        int i=0;
        }
for(i=0;i<NumServers;i++)
if(((Server) servers[i]).ServerID == serverID)
    { ((Server) servers[i]).serviceComplete(currentTime, serverID, aCategory);
        break;
    }
}

public void checkServer(long currentTime, int FromQueueID, AllowedCategoriesEnum aCategory)
{
    int i=0;
    for(i=0;i<NumServers;i++)
        if(((Server) servers[i]).FromQueueID == FromQueueID &&
            ((Server) servers[i]).ServerBusy == false )
        {
            Event DequeueEvent = new Event();
            DequeueEvent.Type = EventTypeEnum.DEQUEUE;
            DequeueEvent.EventTime = currentTime;
            DequeueEvent.Category = aCategory ;
            DequeueEvent.ServerID = ((Server) servers[i]).ServerID;
            DequeueEvent.FromQueueID = FromQueueID;
            eventlist.Dispatch(DequeueEvent);
            break;
        }
}

public AllowedCategoriesEnum GetCategory()
{   return Category ; }

package WaitingQueue;

public class Server extends ServiceComponent
{
    protected EventList eventlist;
    protected Measurement TheMeasurement;
    protected float Productivity;
    protected long MaximumServiceTime;
    public int ServerID;
    public int FromQueueID;
    public boolean ServerBusy;
    public boolean EndOfService;
    public AllowedCategoriesEnum myCategory;
    public Customer CustomerToBeServed;
    public Server(float _Productivity, long _MaximumServiceTime, int _QueueID)
    {
        // Initializing the parameters of the server
        Productivity = _Productivity;
        MaximumServiceTime = _MaximumServiceTime;
        FromQueueID = _QueueID;
        // Initializing the status of the server
        EndOfService = false;
        ServerBusy = false;
    }
}
// Initialize no customer in the Server
CustomerToBeServed = null;
// get the singleton instance
eventlist = EventList.getInstance();
TheMeasurement = Measurement.getInstance();
}

public void EndService()
{
    EndOfService = true;
}

public void serve(Customer aCustomer, long currentTime, int serverID,
    AllowedCategoriesEnum aCategory)
{
    // Those are possibly generated events
    Event ReorderEvent = new Event();
    Event ServiceCompleteEvent = new Event();
    long IntermediateTime;
    // Copy the customer from the event and place it in the server.
    CustomerToBeServed = aCustomer;
    ServerBusy = true;
    // Schedule a REORDER Event
    ReorderEvent.Type = EventTypeEnum.REORDER;
    ReorderEvent.Category = myCategory;
    ReorderEvent.EventTime = currentTime;
    eventlist.Dispatch(ReorderEvent);
    // Schedule a SERVICE_COMPLETE Event
    IntermediateTime = (long) ((CustomerToBeServed.Remaining
        ServiceUnits)\*Productivity);
    if(IntermediateTime< MaximumServiceTime)
    {
        ServiceCompleteEvent.EventTime = currentTime + IntermediateTime;
        CustomerToBeServed.ServiceTime += IntermediateTime;
    }else
    {ServiceCompleteEvent.EventTime = currentTime + MaximumServiceTime;
        CustomerToBeServed.ServiceTime += MaximumServiceTime;
    }
    ServiceCompleteEvent.Type = EventTypeEnum.SERVICE_COMPLETE;
    ServiceCompleteEvent.Category = myCategory;
    ServiceCompleteEvent.ServerID = serverID;
    eventlist.Dispatch(ServiceCompleteEvent);
}

public void serviceComplete(long currentTime, int serverID, AllowedCategoriesEnum
    aCategory)
{
    Event DeactivateOrDequeueEvent = new Event();
    long IntermediateTime;
    ServerBusy = false;
    IntermediateTime = (long)((CustomerToBeServed.Remaining ServiceUnits)\*Productivity);
    if(IntermediateTime> MaximumServiceTime)
    {
        Event ArrivalEvent = new Event();
        IntermediateTime = IntermediateTime - MaximumServiceTime;
        CustomerToBeServed.RemainingServiceUnits = (int)(IntermediateTime/\Productivity);
        // Schedule an arrival event to complete the customer service
        ArrivalEvent.EventTime = currentTime;
        ArrivalEvent.CarriedCustomer = CustomerToBeServed;
        ArrivalEvent.Type= EventTypeEnum.ARRIVAL;
        ArrivalEvent.Category = myCategory;
        eventlist.Dispatch(ArrivalEvent);
    }
}
else // The customer is serviced now
{
    // Accumulate Statistics
    TheMeasurement.AddFinishTime(currentTime);
    TheMeasurement.AddArrivalTime(CustomerToBeServed.ArrivalTime);
    TheMeasurement.AddRequiredServiceUnits(CustomerToBeServed.RequiredServiceUnits);
    TheMeasurement.AddServiceTime(CustomerToBeServed.ServiceTime);
    TheMeasurement.NumCustomers++;
}

if(EndOfService==true)
{
    // Schedule a deactivate event to shutdown this server
    DeactivateOrDequeueEvent.Type=EventTypeEnum.DEACTIVATE_SERVER;
    DeactivateOrDequeueEvent.Category = myCategory;
    DeactivateOrDequeueEvent.ServerID= ServerID;
    DeactivateOrDequeueEvent.EventTime = currentTime;
    eventlist.Dispatch(DeactivateOrDequeueEvent);
}

else
{
    DeactivateOrDequeueEvent.Type= EventTypeEnum.DEQUEUE;
    DeactivateOrDequeueEvent.Category = myCategory;
    DeactivateOrDequeueEvent.FromQueueID = FromQueueID;
    DeactivateOrDequeueEvent.ServerID= ServerID;
    DeactivateOrDequeueEvent.EventTime = currentTime;
    eventlist.Dispatch(DeactivateOrDequeueEvent);
}

}
protected EventList eventlist;
protected QueuingComponent queueCategories[];
protected int MaxNumCategories;
protected int NumCategories;
public QueueFacility(int _MaxNumCategories)
{
    MaxNumCategories = _MaxNumCategories;
    queueCategories = new QueueCategory[MaxNumCategories];
    NumCategories = 0;
    eventlist = EventList.getInstance();
}
public Customer Dequeue(AllowedCategoriesEnum CatID, int QID, long CurrentTime, int ServerID)
{
    Customer ctemp;
    for (int i=0; i < NumCategories; i++)
    {
        if (((QueueCategory) queueCategories[i]).Category == CatID)
        {
            ctemp = ((QueueCategory) queueCategories[i]).Dequeue(CatID, QID, CurrentTime,
                    ServerID);
            if (ctemp != null)
            {   // Schedule a SERVE event to that customer
                Event ServeEvent = new Event();
                ServeEvent.CarriedCustomer = ctemp;
                ServeEvent.Type = EventTypeEnum.SERVE;
                ServeEvent.Category = CatID;
                ServeEvent.ServerID = ServerID;
                ServeEvent.EventTime = CurrentTime;
                eventlist.Dispatch(ServeEvent); return ctemp;
            }
        }
    }
    return null;
}
public void Enqueue(Customer aCustomer)
{
    for (int i=0; i < NumCategories; i++)
    {
        if (((QueueCategory) queueCategories[i]).Category == aCustomer.Category)
        {
            ((QueueCategory) queueCategories[i]).Enqueue(aCustomer);
            break;
        }
    }
}
public void addQueueCategory(AllowedCategoriesEnum CatID, boolean ReorderFlag)
{
    queueCategories[NumCategories] = new QueueCategory(SimGlobals.MaxQueuesInCat,
            CatID);
    ((QueueCategory) queueCategories[NumCategories]).Category = CatID;
    ((QueueCategory) queueCategories[NumCategories]).Reorderable = ReorderFlag;
    NumCategories++;
};
public int addQueue(AllowedCategoriesEnum aCategory)
{
    for (int i=0; i<NumCategories; i++)
    {
        if (((QueueCategory) queueCategories[i]).Category == aCategory)
public void removeQueue(AllowedCategoriesEnum CatID, int QID)
{
    for (int i=0; i < NumCategories; i++)
    {
        if (((QueueCategory) queueCategories[i]).Category == CatID)
        {
            ((QueueCategory) queueCategories[i]).removeQueue(CatID, QID);
            break;
        }
    }
}

public void ReorderCategory(AllowedCategoriesEnum CatID)
{
    for (int i=0; i < NumCategories; i++)
    {
        if (((QueueCategory) queueCategories[i]).Category == CatID)
        {
            ((QueueCategory) queueCategories[i]).Reorder();
            break;
        }
    }
}

public int getNumberOfCategories(){  return NumCategories;  }

package WaitingQueue;
public class QueueCategory extends QueuingComponent
{
    protected EventList eventlist;
    protected QueuingComponent queues[];
    protected int MaxNumQueues;
    protected int NumQueues;
    protected int QueueIDs; //unique IDs for members of this category
    protected AllowedCategoriesEnum Category;
    public boolean Reorderable;

    public QueueCategory(int _MaxNumQueues, AllowedCategoriesEnum _Category)
    {
        Category = _Category;
        MaxNumQueues = _MaxNumQueues;
        queues = new Queue[MaxNumQueues];
        NumQueues = 0;
        QueueIDs = 0;
        Reorderable = false;
        eventlist = EventList.getInstance();
    }

    public int addQueue(AllowedCategoriesEnum aCategory)
    {
        if (NumQueues != MaxNumQueues)
```java
{
    queues[NumQueues] = new Queue(SimGlobals.MaxQueueLength);
    ((Queue) queues[NumQueues]).myCategory = aCategory;
    ((Queue) queues[NumQueues]).QueueID = QueueIDs;
    QueueIDs++;
    NumQueues++;
    return QueueIDs-1;
}
return -1;
}

public Customer Dequeue(AllowedCategoriesEnum CatID, int queueID, long currentTime, int serverID)
{
    int i=0;
    Customer dequeuedCustomer = null;
    for(i=0;i<NumQueues;i++)
        if( Category==CatID && ((Queue) queues[i]).QueueID == queueID && ((Queue) queues[i]).getLengthOf() != 0 )
        {
            dequeuedCustomer = ((Queue) queues[i]).Dequeue( CatID, queueID, currentTime, serverID);
            break;
        }
    return dequeuedCustomer;
}

public void Enqueue(Customer aCustomer)
{
    int min = 0;
    // Select the queue with the minimum number of customers lining up.
    for (int i=1; i < NumQueues; i++)
    {
        if (((Queue) queues[i]).getLengthOf() < ((Queue) queues[min]).getLengthOf())
        {
            min = i;
        }
    }
    queues[min].Enqueue(aCustomer);
    if(((Queue) queues[min]).getLengthOf() == 1)
    {
        Event CheckServerEvent = new Event();
        CheckServerEvent.Type = EventTypeEnum.CHECK_SERVER;
        CheckServerEvent.EventTime = aCustomer.ArrivalTime;
        CheckServerEvent.Category = Category;
        CheckServerEvent.FromQueueID = ((Queue) queues[min]).QueueID;
        eventlist.Dispatch(CheckServerEvent);
    }
}

public void Reorder()
{
    Customer c;
    int min = 0;
    int max = 0;
    for (int i=1; i < NumQueues; i++)
    {
        if (((Queue) queues[i]).getLengthOf() < ((Queue) queues[min]).getLengthOf())
        {
            min = i;
        }
        if (((Queue) queues[i]).getLengthOf() > ((Queue) queues[max]).getLengthOf())
        {
            max = i;
        }
    }
}
```
while (((Queue) queues[max]).getLengthOf() - ((Queue) queues[min]).getLengthOf() > 1)
{
    c = ((Queue) queues[max]).removeLast();
    ((Queue) queues[min]).Enqueue (c);
    min = 0;
    max = 0;
    for (int i=1; i < NumQueues; i++)
    {
        if (((Queue) queues[i]).getLengthOf() < ((Queue) queues[min]).getLengthOf()) {
            min = i;
        }
        if (((Queue) queues[i]).getLengthOf() > ((Queue) queues[max]).getLengthOf()) {
            max = i;
        }
    }
}

package WaitingQueue;

public class Queue extends QueuingComponent {
    Customer[] contents;
    int front;
    int length;
    int MaxLength;
    public AllowedCategoriesEnum myCategory;
    public int QueueID;

    public Queue(int maxQueueLength)
    {
        contents = new Customer[maxQueueLength];
        front = 0;
        length = 0;
        MaxLength = maxQueueLength;
    }

    public Customer removeLast()
    {
        Customer lastCustomer;
        lastCustomer = contents [(front+length)%MaxLength - 1];
        length—;
        return lastCustomer;
    }

    public int getLengthOf (){return length;
    }

    public int getMaxLength (){ return MaxLength; }

    public void Copy (Queue cq)
    {
        Customer ctemp;
        int len = getLengthOf();
        for (int i = 0; i < len; i++)
        {
            }
ctemp = Dequeue(myCategory, QueueID, 0,0);
cq.Enqueue(ctemp);
}
cq.QueueID = QueueID;
}
public void Enqueue(Customer aCustomer)
{
    contents [(front+length)%MaxLength] = aCustomer;
    length++;
}
public Customer Dequeue(AllowedCategoriesEnum CatID, int QID, long CurrentTime, int ServerID)
{
    Customer currentCustomer;
    currentCustomer = contents [front];
    front = (++front)%MaxLength;
    length--;
    return currentCustomer;
}

/***********************************
*                                *
*      AbstractGenerator.jav      *
*                                *
***********************************/

package WaitingQueue;

public abstract class AbstractGenerator
{
    protected EventList eventlist = EventList.getInstance();
    public void generateArrival(long currentTime, Customer new_customer)
    {
        Event new_event = new Event();
        new_event.Type = EventTypeEnum.ARRIVAL;
        // Set the time of the next arrival
        long TimeIncrement = ArrivalTimePolicy(currentTime);
        new_event.EventTime = currentTime + TimeIncrement;
        new_customer.ArrivalTime = new_event.EventTime;
        // Set the type of the customer
        AllowedCategoriesEnum customerCategory = ArrivalCategoryPolicy();
        new_event.Category = customerCategory;
        new_customer.Category = customerCategory;
        // Set the service units for the customer
        int serviceUnits = ServiceUnitsPolicy();
        new_customer.RequiredServiceUnits = serviceUnits;
        new_customer.RemainingServiceUnits = new_customer.RequiredServiceUnits;
        // Add the event to the event list handler
        new_event.CarriedCustomer = new_customer;
        eventlist.Dispatch(new_event);
    }

    public abstract long ArrivalTimePolicy(long currentTime);
    public abstract AllowedCategoriesEnum ArrivalCategoryPolicy();
    public abstract int ServiceUnitsPolicy();
}

/***********************************
*                                *
*                                *
*                                *
***********************************
package WaitingQueue;
public class RandomArrivals extends AbstractGenerator
{
    public long ArrivalTimePolicy(long currentTime)
    {
        double randomNumber = Math.random();
        return (long) (randomNumber * (double) 5);
    }
    public AllowedCategoriesEnum ArrivalCategoryPolicy()
    {
        double randomNumber = Math.random();
        if(randomNumber < 0.5)
            return AllowedCategoriesEnum.COACH;
        else return AllowedCategoriesEnum.BUSINESS;
    }
    public int ServiceUnitsPolicy()
    {
        double randomNumber = Math.random();
        return (int) (randomNumber * (double) 15);
    }
}

package WaitingQueue;
public class EventList
{
    private event_node head_event;
    static private EventList theInstance = null;
    static public EventList getInstance()
    {
        if(null == theInstance)
        {
            theInstance = new EventList();
        }
        return theInstance;
    }
    protected EventList() { head_event = null; }
    public void clear()
    {
        event_node top_node;
        while( head_event != null) // Repeat till the list is empty
        {
            head_event.ptr_event_object = null;
            top_node = head_event;
            head_event = head_event.next_event_node;
            top_node = null;
        }
    }
    public void Dispatch(Event new_event)
    {
        // If the event list is empty , then it is the first event in the
        // list , add a new node and make it the head of the list
        if(head_event == null)
        {
            head_event = new event_node();
        }
    }
}
head_event.ptr_event_object = new_event;
head_event.next_event_node = null;
return;
}
// There is already events in the list, then search for the
// place to add the event according to the time of the event to
// be added compared with the time of the events already in the
// existing nodes.
event_node current = head_event;

// check if the node we want to add will be the first node in the
// event list, because its time is less than the time of the
// head node.
if (new_event.EventTime <= head_event.ptr_event_object.EventTime)
{
    event_node new_node = new event_node();
    new_node.ptr_event_object = new_event;
    new_node.next_event_node = head_event;
    head_event = new_node;
    return;
}
// Search until we reach the end of the event list
while(current.next_event_node != null)
{
    if (new_event.EventTime <= (current.next_event_node).ptr_event_object.EventTime)
    {
        // Add the new node with the new event between the
        // current and the next node
        event_node new_node = new event_node();
        new_node.ptr_event_object = new_event;
        new_node.next_event_node = current.next_event_node;
        current.next_event_node = new_node;
        return;
    }
    current = current.next_event_node;
}
// We searched the list and we did not find an event whose time
// is larger than the time of the event we want to schedule, that
// means the event we want to schedule is the last event we will
// add to the list
current.next_event_node = new event_node();
current = current.next_event_node;
current.ptr_event_object = new_event;
current.next_event_node = null;
}
public Event GetNextEvent()
{
    // If there is no events in the event_list queue, return NULL
    if (head_event == null) return null;
    // Create a pointer to point to the event to be returned
    Event event_to_dispatch = head_event.ptr_event_object;
    // Create a node pointer to point to the node to be deleted from the
    // event list
    event_node top_node = head_event;
    // Make the next node to be the head node
    head_event = head_event.next_event_node;
    // Delete the top_node
    top_node = null;
    // return the required event to be simulated
    return event_to_dispatch;
}
package WaitingQueue;
public class event_node{
    public event_node next_event_node;
    public Event ptr_event_object;
    public event_node()
    {
        next_event_node = null;
        ptr_event_object = null;
    }
}

package WaitingQueue;
public class Event{
    public long EventTime;
    public EventTypeEnum Type;
    public AllowedCategoriesEnum Category;
    public Customer CarriedCustomer;
    public int FromQueueID;
    public int ServerID;
    public Event(){ CarriedCustomer = null; }
}

package WaitingQueue;
import java.io.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;
import javax.print.event.*;
public class EventTypeEnum{
    public static final EventTypeEnum ARRIVAL = new EventTypeEnum();
    public static final EventTypeEnum REORDER = new EventTypeEnum();
    public static final EventTypeEnum DEQUEUE = new EventTypeEnum();
    public static final EventTypeEnum SERVE = new EventTypeEnum();
    public static final EventTypeEnum SERVICE_COMPLETE = new EventTypeEnum();
    public static final EventTypeEnum CHECK_SERVER = new EventTypeEnum();
    public static final EventTypeEnum ACTIVATE_SERVER = new EventTypeEnum();
    public static final EventTypeEnum DEACTIVATE_SERVER = new EventTypeEnum();
    private static final String[] stringTable = {
        "ARRIVAL",  ...
"REORDER",
"DEQUEUE",
"SERVE",
"SERVICE_COMPLETE",
"CHECK_SERVER",
"ACTIVATE_SERVER",
"DEACTIVATE_SERVER"
];
protected String[] getStringTable() {
    return stringTable;
}
private static final EventTypeEnum[] enumValueTable = {
    ARRIVAL,
    REORDER,
    DEQUEUE,
    SERVE,
    SERVICE_COMPLETE,
    CHECK_SERVER,
    ACTIVATE_SERVER,
    DEACTIVATE_SERVER
};
protected EventTypeEnum[] getEnumValueTable() {
    return enumValueTable;
}

public class AllowedCategoriesEnum {
    public static final AllowedCategoriesEnum COACH = new AllowedCategoriesEnum();
    public static final AllowedCategoriesEnum BUSINESS = new AllowedCategoriesEnum();
    private static final String[] stringTable = {
        "COACH",
        "BUSINESS"
    };
    protected String[] getStringTable() {
        return stringTable;
    }
    private static final AllowedCategoriesEnum[] enumValueTable = {
        COACH,
        BUSINESS
    };
    protected AllowedCategoriesEnum[] getEnumValueTable() {
        return enumValueTable;
    }
}

/package WaitingQueue/
import java.io.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;
import javax.print.event.*;
public class AllowedCategoriesEnum {
    public static final AllowedCategoriesEnum COACH = new AllowedCategoriesEnum();
    public static final AllowedCategoriesEnum BUSINESS = new AllowedCategoriesEnum();
    private static final String[] stringTable = {
        "COACH",
        "BUSINESS"
    };
    protected String[] getStringTable() {
        return stringTable;
    }
    private static final AllowedCategoriesEnum[] enumValueTable = {
        COACH,
        BUSINESS
    };
    protected AllowedCategoriesEnum[] getEnumValueTable() {
        return enumValueTable;
    }
}
package WaitingQueue;
public class Customer
{
    public long ArrivalTime;
    public long ServiceTime;
    public int RequiredServiceUnits;
    public int Priority;
    public AllowedCategoriesEnum Category;
    public long FinishTime;
    public int RemainingServiceUnits;
    public Customer()
    {
    }
    public Customer(long CurrentTime, int RequiredUnits, AllowedCategoriesEnum _Category)
    {
        FinishTime = 0;
        ArrivalTime = CurrentTime;
        RequiredServiceUnits = RequiredUnits;
        RemainingServiceUnits = RequiredUnits;
        ServiceTime = 0;
        Category = _Category;
        Priority = 0;
    }
}

package WaitingQueue;
public class Measurement
{
    public int NumCustomers;
    public long ArrivalTime[];
    public long FinishTime[];
    public long RequiredServiceUnits[];
    public long ServiceTime[];
    static private Measurement theInstance = null;
    static public Measurement getInstance()
    {
        if(null == theInstance) {
            theInstance = new Measurement();
        }
        return theInstance;
    }
    protected Measurement()
    {
        NumCustomers = 0;
        ArrivalTime = new long[SimGlobals.MaxNumOfCustomers];
        FinishTime = new long[SimGlobals.MaxNumOfCustomers];
        RequiredServiceUnits = new long[SimGlobals.MaxNumOfCustomers];
        ServiceTime = new long[SimGlobals.MaxNumOfCustomers];
    }
    public long AverageWaitingTime()
    {
        long Average = 0;
        for(int i = 0; i < NumCustomers; i++)
Average += (FinishTime[i]-ArrivalTime[i] - ServiceTime[i]);
Average = Average/NumCustomers;
return Average;
}

public double StdWaitingTime()
{
    long Average=0;
    Average = AverageWaitingTime();
    double Std=0;
    long Diff=0;
    for(int i=0;i<NumCustomers;i++)
    {
        Diff = (FinishTime[i]-ArrivalTime[i]-ServiceTime[i]-Average);
        Diff = Diff*Diff;
        Std += Diff;
    }
    Std = Std/(NumCustomers-1);
    Std = Math.sqrt(Std);
    return Std;
}

public long MaxWaitingTime()
{
    long Max=0;
    for(int i=0;i<NumCustomers;i++)
    if((FinishTime[i]-ArrivalTime[i]-ServiceTime[i]) > Max) Max = (FinishTime[i]-ArrivalTime[i]-ServiceTime[i]) ;
    return Max;
}

public void AddArrivalTime(long arrival) { ArrivalTime[NumCustomers] = arrival;};
public void AddFinishTime(long finish) { FinishTime[NumCustomers] = finish;};
public void AddServiceTime(long Service) { ServiceTime[NumCustomers] = Service;};
public void AddRequiredServiceUnits(int service) { RequiredServiceUnits[NumCustomers] = service;};

/*****************************/
* SimGlobals.java
*
/*****************************/

package WaitingQueue;
public class SimGlobals
{
    public long SimTime;
    public EventTypeEnum EventType;
    public static final int MaxNumOfCustomers = 1000 ;
    public AllowedCategoriesEnum AllowedCategories;
    public static final int MaxQueueLength = 10 ;
    public static final int MaxCategoriesInFac = 2 ;
    public static final int MaxQueuesInCat = 1 ;
}

Figure 12-13. A UML class diagram model for the check-in counters example.
Summary

This chapter discusses the applicability of the POAD process to develop a pattern-oriented design for the domain of waiting-queue simulation. The example is simple, yet illustrative. The final class diagram is instantiable for all applications that fall within the boundaries of the same domain, as discussed in the introduction of this chapter. In the following chapter we illustrate a digital content-remastering system.
Chapter 13. A Digital Content Remastering Application

Document Understanding

Pattern-Oriented Analysis and Design for the Distribution Subsystem

Pattern-Oriented Analysis and Design for the Filtering Subsystem

Summary
Document Understanding

The design of many software systems often involves the manipulation and processing of digital media or digital content. For instance, the ability to deliver electronic services through internet-based delivery channels requires that printed material such as books, journals, newspapers, and magazines be converted into forms suitable for electronic distribution. This type of content manipulation often includes preprocessing, transformation from one format to another, extraction of metadata, and in many cases verification and validation of the resulting content.

[1] We use the terms digital media, digital content, and data interchangeably.

Document understanding is one form of content understanding in which a system analyzes documents, including books, journals, and enterprise corporate documents. Document understanding as a field is concerned with the semantic analysis of documents to extract human-understandable information and codify it into machine-readable and machine-understandable information.

There is a huge effort expended on the automation of document-understanding activities. The result of such work is the development of document understanding algorithms and applications and systems utilizing document-understanding techniques. In this chapter we focus on the design of document-understanding applications and systems using the POAD approach.

Applications and Systems

Document understanding systems provide automated means to extract meaningful information from a raster image of a document. Those systems provide engines for content remastering and content repurposing for delivery through different output channels. They provide means to create information-rich content that is usable in many end-user applications, such as search and retrieval. To process a mass amount of data, such as a collection of books and journals produced by a publisher, content-understanding systems should run nonstop in an automated fashion and in an unattended operation mode.

A document-understanding system often starts with a scanned image representation of a document and delivers an information-rich representation together with useful semantic information about its content. For instance, input channels could feed the system with a raster image format of document pages (such as the TIFF or BMP formats); the output channel could deliver searchable Portable Document Format (PDF) or eXtensible Markup Language (XML) documents. One application of such a system could be the remastering of digitized (image) documents and repurposing the content to become the backend of an online web community. A practical example of a community that uses a backend of repurposed material is the Cognitive Science Online Community.

A document-understanding system has a large number of components that work together in one or several workflows to achieve the system-level functionality. These components include, for example, algorithms for generation of digitized text, layout analysis, extraction of logical information, text understanding, corpus-wide metadata extraction, and journal and chapter splitting. These algorithms are developed in-house, acquired commercially, or obtained off-the-shelf as freeware or shareware components. The system uses this collection of algorithms to provide various functionalities.

Today, several document-understanding applications exist, either for research purposes or for commercial use. These applications perform document-understanding functionalities and contain several document-understanding algorithms. For instance, the WISDOM++ application is an intelligent document-processing system that transforms paper documents into digital format such as HTML or XML. The system has several components for OCR, segmentation, classification, layout analysis, and transformation into web-accessible format. Roussel, Hitz, and Ingold (2001) identify the requirements for a document-understanding system that emphasizes the user-user interaction, user-algorithm interaction, and algorithm-algorithm interaction. Other products, such as Abbyy FineReader and Adobe Acrobat Capture, are considered applications for transforming a raster image document into a searchable structured format like PDF or XML.

Existing solutions are considered document-understanding applications or programs; they are not document understanding systems.
Applications are useful for end users, since they provide graphical user interfaces through which the user interacts with the system, accepts or rejects results, or resubmits the document for processing with alternative configurations. We call these solutions standalone applications.

Standalone applications are not suitable for processing massive volumes of data and large numbers of document pages. This is often the case for commercial publishing. For instance, a publisher might want to provide online web access to a collection of books and journals that belong to some domain through a web community or through the publisher's web site. For this purpose, this collection of books and journals must be converted from paper format into searchable digital formats such as PDF documents. Using standalone applications to convert the collection is not a feasible solution due to the enormous manpower required to use these applications to convert each and every document page in the collection.

As an example, consider a publisher who wants to convert the out-of-print collection from paper format to digital searchable format for the purpose of providing a print-on-demand (POD) service of those out-of-print items for his customers. The publisher has an out-of-print collection of 5,000 books and journals. The document images produced from scanning this collection (which is a moderate-sizes collection) will be around 15 terabytes of data, or approximately 1.5 million pages. The estimated size of the output data in PDF format would be in the 200 Gigabytes range depending on the content of the pages. With such a large number of pages, a standalone application is not suitable.

To process such a massive amount of data, we need an automated system that has several key features:

1. The system runs in an unattended operation mode. Human intervention with the system should be minimized.
2. The system runs in a nonstop operation mode. In order to be able to process this massive amount of data in a reasonable timeframe, the system should run in a 24/7 operation mode.
3. The system operates on multiple documents at the same time, using a cluster of processing nodes that collaborate to process the corpus of documents.

In the following section, we discuss an example of a system that is developed for this purpose, the Digital Content Remastering (DCRM) system.

A DCRM System

The DCRM system is a complex system that provides automated, nonstop unattended document-understanding operations. Complexity arises from the number of software and hardware components that are integrated together as a solution for document understanding. The system has components that implement algorithms for document understanding. The system also has components for managing the operation and providing important nonfunctional features such as reliability and fault tolerance. The complexity also arises from management of a large volume of input data as well as management of the output data. The data flow and control flow in such a system is complex, and the architecture should provide means to manage this complexity and facilitate interaction between components. Figure 13-1 illustrates an example of the hardware setup for a DCRM system.
The system is composed of a cluster of workstations that are connected using a network switch. The workstations are called worker machines (or mill machines). Each machine is capable of executing components that perform one or more document-understanding algorithms. The worker machines are connected to a main server that hosts an external storage. The main server is responsible for managing the input data. There are several input channels to supply the system with scanned document images. In Figure 13-1, we illustrate two input channels: a standalone tape drive and a tape library. Both input channels can read Digital Linear Tapes (DLTs); each tape may contain approximately 25 to 30 books. A RAID (redundant arrays of independent disk drives) storage is used to hold the input data, while the worker machines are performing the document understanding operations. The output data is stored on a separate server. A delivery server is used to collect output data and write them to a delivery media such as DLT tapes or compact discs (CDs).

The architecture of the system is based on component-based software engineering (CBSE) principles [Szyperski 1998; CBSE 2001]. The architecture is a composition of components using a glue framework. Components perform the document-understanding functions, such as OCR, layout analysis, and logical structure analysis. Each component is a self-contained, fairly independent, concrete realization that provides well-defined services and functionalities. Components are glued together using glue infrastructure that manages the workflow execution.

The system has many components, including the following:

- **Content understanding components** such as OCR engines, journal-splitting components, page layout analysis components, logical analysis components, PDF concatenation component, and journal article–creation component.
- **Access channel components** to manage tape libraries, standalone tapes, and other entry-point channels for the system.
- **Quality assurance components** such as the input data-integrity verifier and automated quality assurance of the output.
- **Preprocessing components** such as bleed through removal and auto-exposure components.
- **Delivery channels components** such as DLT writers and CD writer components.
- **Configuration utilities** such as tools to submit new titles to be processed or remove titles from the processing queues and tools to check the processing status of the corpus.
- **System components** such as reliable wire transfer, logger, watchdogs, process monitors, release synchronizers, event reporters, and performance-analysis components.

To integrate this variety of components, the architecture is built using the software framework principles, mainly the "don't call us, we'll call you" concept. In this architecture, none of the components has an explicit knowledge of the existence of other components in the system, whether they are on the same worker machine or on any other machine. Processes run in parallel over a distributed cluster of workstations. The data and control flows are separated in the architecture. The data flow is managed using the external RAID system. The control flow is managed by a centralized engine that uses a set of communication channels. Each process knows of an input channel to read tasks from and produce output in either success channels that carry successful tasks or fail channels that carry failed tasks.
It is not the objective of this chapter to describe the details of the DCRM system. The system is too complex and large to discuss its design in one chapter. Instead, we focus on two subsystems for illustration purposes.

**DCRM Subsystems**

The DCRM system is composed of several subsystems. From our earlier discussions we can easily identify some of these subsystems.

First, we need a subsystem that deals with the distribution of tasks across the worker machines; we call this the *distribution* subsystem. For example, the task in this case could be the application of the required document-analysis algorithms to convert a page from TIFF format to PDF format. The worker machines will be working in parallel, each processing a conversion task.

We can also recognize the need for a subsystem that handles the application of several content-understanding algorithms to a page document to convert it from the TIFF to PDF format. The application of those algorithms is handled through a subsystem for data processing and filtering; we call this the *filtering* subsystem.

Another example of the DCRM subsystem is the controller engine that handles the workflow logic and masters the execution of components in the system. We call it the *controller* subsystem.

In this chapter we do not address the analysis and design of the whole DCRM system. We have selected two subsystems to discuss: the distribution subsystem and the filtering subsystem.
Pattern-Oriented Analysis and Design for the Distribution Subsystem

In this section, we use the POAD process to analyze and design the Distribution subsystem of the DCRM application.

POAD Analysis for the DCRM Distribution Subsystem

Requirements Analysis

We now analyze the first subsystem, the distribution subsystem. The distribution subsystem resides on the main server and is responsible for distributing tasks to other worker machines in the cluster. It hides all the details about communicating with a worker machine. We have chosen the name distribution subsystem over a communication subsystem because this subsystem provides functionalities other than just communicating messages between the main server and the worker machines.

As part of the requirements for this subsystem, the distribution subsystem should provide the following functionalities:

1. Provide an interface that accepts task descriptions from the controller engine and receives notification messages from the worker machines regarding the completion of a task. This interface provides the necessary transparency such that the rest of the application does not have to know where a task is executed or how it is communicated to the worker machine.

2. Provide the necessary communication mechanisms to communicate with all worker machines. Therefore, the distribution subsystem hides the implementation of the communication protocol from the rest of the application.

3. Be extensible to support several communication technologies and protocols without affecting the implementation of the rest of the application.

4. Be able to communicate with heterogeneous worker machines. The worker machines do not have to use the same communication technologies. Therefore, the distribution subsystem should be able to provide an interface to each worker machine whose implementation depends on the technology supported by that particular worker machine. This heterogeneity of the communication protocol should be transparent to the application.

5. Provide a mechanism for distributing tasks to worker machines. The rest of the application does not know how a task is distributed to a worker machine; it is the responsibility of the distribution subsystem to select the machine to be used for performing the submitted task.

6. Be extensible in terms of the number of machines it supports, the technologies used for communicating with worker machines and the policy used to select a machine to which a task will be assigned.

Acquaintance and Retrieval

From the requirements of the system, it is clear that the subsystem is related to two main fields: distributed systems and communication systems. The analyst must become acquainted with design patterns in this field that could be useful in the design of the distribution subsystem.

There is a variety of patterns for concurrent and networked objects discussed in volume 2 of Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects [Schmidt et al., 2000]. In addition, the literature is full of patterns for distributed and
communication systems. Some candidates include the Proxy pattern [Gamma et al. 1995], the Acceptor pattern [Schmidt 1998b], the Active Object pattern [Lavender & Schmidt 1999], the Client Dispatcher Server pattern [Buschmann et al. 1996; Sommerlad & Stal 1996], the distributed processes patterns [Shaw & Garlan 1996], the generative pattern language for distributed processing [DeBruler 1995], the pattern system for network management interfaces [Keller & Schauer 1998], the Reactor pattern [Schmidt 1995], and the Proactor pattern [Pyarali et al. 1999].

These patterns, among others, provide a rich collection of patterns to choose from when designing the distribution subsystem of the DCRM application.

Pattern Selection

Based on the requirements in earlier steps, we select the patterns that could be used in the design of the distribution subsystem. We have decided to simplify the design of this subsystem by using versatile design patterns as follows.

To provide the necessary transparency, the distribution subsystem will communicate with a representative for each worker machine rather than directly with the worker machines. Therefore, we need a local representative (or agent) to represent the remote machine. The actual reference to the remote machine is kept in the local representative, and the rest of the system communicates with this representative. A Proxy pattern [Gamma et al. 1995] provides a proxy object that acts as a surrogate or placeholder for another object. In our case the proxy plays the role of a local representative to the remote object, which is the remote worker machine.

The distribution subsystem manages the process of selecting the worker machine to which a new task is assigned. It should provide several possible mechanisms for selecting the worker machine. Therefore, the subsystem should define a family of strategies for selecting a worker machine to which a task is delegated. The subsystem encapsulates those strategies and makes them interchangeable. The Strategy pattern [Gamma et al. 1995] is used to encapsulate a set of selection strategies and to allow the strategies to vary independently of the rest of the application using the distribution subsystem.

The distribution subsystem manages the distribution among several worker machines in the cluster. The subsystem keeps a proxy object for each worker machine it communicates with. Based on the nature of the worker machine, it may be desirable to communicate with a particular machine using a particular communication protocol. Therefore, we end up with a variety of proxy types. The distribution subsystem should manage the creation of this variety of proxies and should communicate with each using a common interface. To deal with this heterogeneity, we use an AbstractFactory pattern [Gamma et al. 1995] to provide an interface for creating families of related objects (proxies) without specifying their concrete classes.

Therefore, for the design of the distribution subsystem, we select three patterns: a Proxy pattern to provide a local representative to remote machines, a Strategy pattern to provide various strategies (policies) for selecting the machine to which a task is assigned, and an AbstractFactory pattern to manage the creation of proxies of different types for different worker machines.

POAD Design for the DCRM Distribution Subsystem

Constructing the Pattern-Level Diagram

In this step, we create instances of the selected patterns and identify the relationships between these instances. As a result, a Pattern-Level diagram of the distribution subsystem is developed.

In the analysis phase we selected three patterns: Proxy, AbstractFactory, and Strategy. We create a pattern instance MachineProxy of type Proxy pattern to implement the surrogate of a remote machine. We create a pattern instance TaskDistribution of type Strategy to provide the necessary encapsulation for the policy used to select the machine to which a task is assigned. We also create a pattern instance ProxyFactory of type AbstractFactory to manage the creation of proxy objects for various worker machines.
We then define the dependency relationships between the pattern instances. The TaskDistribution pattern instance uses the ProxyFactory pattern instance to create the proxy objects for all the worker machines. The ProxyFactory pattern instance creates the proxies in the MachineProxy pattern instance. The TaskDistribution pattern instance then uses the MachineProxy pattern instance to execute tasks on remote machines. Figure 13-2 illustrates the Pattern-Level diagram for the distribution subsystem.

**Figure 13-2. The Pattern-Level diagram for the DCRM distribution subsystem.**

![Diagram](image)

**Constructing the Pattern-Level with Interfaces Diagram**

In this step we analyze the relationships between pattern instances. The dependency relationship between patterns in the Pattern-Level view is a conceptual high-level uses relationship. We further trace these dependencies to lower-level design.

We first declare the interfaces for the patterns used in the Pattern-Level diagram. The interface for the Strategy pattern is the Context interface class behind which the distribution policy is encapsulated. The AbstractFactory pattern has two interfaces. The first interface is for the factory object AbstractFactory interface class, which provides an interface for creating various types of products. The second interface is for the product created by the factory, the AbstractProduct interface class, which provides an interface for all various products created by the factory (the proxies in this case study). The Proxy pattern has one interface, the Subject interface class, which defines the common interface for the real subject (the remote machine) and the proxy (the local representative).

We then translate the dependency relationships between pattern instances in Figure 13-2 into pattern interface relationships. The relationship between the pattern instances TaskDistribution and ProxyFactory in Figure 13-2 is then translated into relationship between their interfaces: the Context interface class and the AbstractFactory interface class. The relationship between the TaskDistribution and the MachineProxy pattern instances is translated into a relationship between their interface classes, Context and Subject. Similarly, the relationship between the pattern instances ProxyFactory and the MachineProxy is translated into a relationship between their interface classes, AbstractProduct and Subject. Figure 13-3 illustrates the Pattern-Level with Interfaces diagram for the DCRM distribution subsystem.

**Figure 13-3. The Pattern-Level with Interface for the DCRM distribution subsystem.**

![Diagram](image)
Constructing the Detailed Pattern-Level diagrams

To construct the Detailed Pattern-Level diagram for the distribution subsystem, we express the internal parts of each instantiated pattern from the Pattern-Level with Interfaces diagram in Figure 13-3. The structure of the three pattern instances MachineProxy of type Proxy, TaskDistribution of type Strategy, and ProxyFactory of type AbstractFactory can be found in Design Patterns: Elements of Object-Oriented Software [Gamma et al. 1995]. Figure 13-4 illustrates the Detailed Pattern-Level diagram.

Figure 13-4. The Detailed Pattern-Level diagram for the DCRM distribution subsystem.
POAD Design Refinement for the DCRM Distribution Subsystem

Instantiating Pattern Internals

In this step we add application-specific nature to the Detailed Pattern-Level diagram by renaming internal pattern classes according to the application design environment, choosing names for pattern participants that are meaningful in the application context, and defining application specific names for operations in patterns.

Instantiating the internals of the TaskDistribution is illustrated in Figure 13-5. The TaskDistribution pattern instance is composed of the following:

- **SchedulerPolicy.** The Strategy interface class of the Strategy pattern is renamed SchedulerPolicy. It provides the interface for selecting a machine proxy to which a task will be delegated. The mechanism by which a machine proxy is selected is implemented in concrete classes that provide a concrete implementation for the interface operation SelectMachine().

- **DistributionContext.** The Context class of the Strategy pattern is renamed DistributionContext, in which the PerformTask operation replaces the contextInterface() operation.

- **RoundRobin and BalancedPerformance.** These concrete classes provide implementation to the SchedulerPolicy interface class by providing a policy for selecting a machine to which a task will be assigned.

**Figure 13-5. Instantiation of the TaskDistribution pattern.**

Instantiating the internals of the MachineProxy is illustrated in Figure 13-6. The MachineProxy pattern instance is composed of:

- **ProxyIF.** The Subject interface class of the Proxy pattern is renamed ProxyIF. ProxyIF plays the role of a common interface for all the proxy classes and the real objects being referred to by the proxy classes.

- **LocalProxy.** LocalProxy is a concrete realization of the ProxyIF. It plays the role of aProxy class in the Proxy pattern.
RemoteSubject. The RealSubject in the Proxy pattern is renamed to RemoteSubject class. RemoteSubject is the concrete subject referred to by the LocalProxy and implements the ProxyIF interface.

**Figure 13-6. Instantiation of the MachineProxy pattern.**

Instantiating the internals of the ProxyFactory is illustrated in **Figure 13-7**. The ProxyFactory pattern instance is composed of the following:

- **Machine1Proxy.** Machine1Proxy plays the role of the ProductA in the AbstractFactory pattern. It defines an interface for a proxy for machine number one in the cluster. Similarly, Machine2Proxy defines an interface for machine number two, and so on.

- **ProxyFactory.** The AbstractFactory class of the AbstractFactory pattern is renamed to ProxyFactory. ProxyFactory defines an interface for creating the proxy objects for each machine in the cluster.

- **RMI-Factory and CORBA-Factory.** RMI-Factory and CORBA-Factory are concrete implementations of the ProxyFactory interface. The RMI-Factory creates a proxy object for each machine that works with Remote Method Invocation (RMI) technology. Similarly, the CORBA-Factory creates a proxy object for each machine that works with CORBA IIOP technology. The user can also define hybrid factories that create an RMI proxy for one machine and a CORBA proxy for another.

**Figure 13-7. Instantiation of the ProxyFactory pattern.**
Developing an Initial Class Diagram

Starting from the Detailed Pattern-Level diagram in Figure 13-4, we use pattern interfaces and the instantiated details of pattern internals (as obtained in the previous subsection) to construct a UML class diagram. We trace the pattern interface relationships in the Detailed Pattern-Level diagram to relationships between internal classes of the instantiated patterns.

We trace the interface relationship between the Context interface class in the TaskDistribution pattern instance and the Subject interface class of the MachineProxy pattern instance in Figure 13-4. The Context interface class is renamed DistributionContext, as illustrated in Figure 13-5. The Subject interface class is renamed ProxyIF, as illustrated in Figure 13-6. Therefore, the interface relationship is translated into association relationship between the DistributionContext class and the ProxyIF class.

We trace the interface relationship between the Context interface class in the TaskDistribution pattern instance and the AbstractFactory interface class of the ProxyFactory pattern instance shown in Figure 13-4. The AbstractFactory interface class is renamed ProxyFactory, as illustrated in Figure 13-7. Therefore, the interface relationship between Context and AbstractFactory is translated into an association relationship between the DistributionContext class and the ProxyFactory class.

We trace the interface relationship between the AbstractProduct interface class of the ProxyFactory pattern instance and the Subject interface class of the MachineProxy pattern instance shown in Figure 13-4. The AbstractProduct interface class is renamed Machine1Proxy and Machine2Proxy classes, as illustrated in Figure 13-7. The Subject interface class is renamed ProxyIF, as illustrated in Figure 13-6. Therefore, the interface relationship is translated to a class association relationship between the Machine1Proxy and Machine2Proxy interface classes and the ProxyIF interface class.

Figure 13-8 illustrates the initial class diagram for the distribution subsystem after tracing all pattern interfaces to internal design classes of the pattern instances.

Figure 13-8. The initial class diagram of the DCRM distribution subsystem.
It can be easily recognized that the patterns are still notable in the class diagram, as shown by the dotted boxes around the classes. Recall that we do not discard earlier diagrams. As part of POAD, all the models in figures 13-2 through 13-8 are saved as analysis and design models. It is the role of a tool support to save these models and provide the necessary traceability mechanisms between them.

Design Optimization

In this step we inspect the initial class diagram for possible optimization using grouping or reduction operations. For this subsystem, we used only one pattern instance of each pattern type. Therefore, there is no possible optimization by reducing abstract classes.

We then look for classes with related functionalities. We realize that the products produced by the ProxyFactory pattern instance of type AbstractFactory are the proxy classes that will be used by the DistributionContext to send and receive messages from the remote machines. Therefore, the Machine1Proxy and the Machine2Proxy should comply with the ProxyIF interface, and the LocalProxy concrete classes are the concrete classes implementing the Machine1Proxy and the Machine2Proxy classes (Machine1-RMI, Machine1-CORBA, Machine2-RMI, Machine2-CORBA, etc.). We refine the design diagram in Figure 13-8 to make Machine1Proxy and the Machine2Proxy inherit from the interface ProxyIF. We then need to create a RemoteSubject class for each LocalProxy class (as defined in the Proxy pattern). Therefore, we create the classes RemoteMachine1-RMI, RemoteMachine1-CORBA, RemoteMachine2-RMI, and RemoteMachine2-CORBA as examples for RemoteSubject classes. All classes will have to implement the PerformTask() operation of the ProxyIF interface. The refined class diagram for the design is illustrated in Figure 13-9.

Figure 13-9. The refined class diagram for the DCRM distribution subsystem.
The designer then refines the class diagram by adding other method calls and developing the detailed interaction diagram between components.
Pattern-Oriented Analysis and Design for the Filtering Subsystem

In this section, we use the POAD process to analyze and design the filtering subsystem of the DCRM application.

POAD Analysis for the DCRM Filtering Subsystem

Requirements Analysis

A digital content-remastering application involves the manipulation and processing of digital content. This type of operation often includes preprocessing, transformation, formatting, data extraction, and various content-understanding functionalities. A digital content-remastering application can be thought of as the integration, composition, and cascade of processing modules or units. We call these processing modules filters, and hence the name of the subsystem responsible for these filtering functionalities. Each filter manipulates input data and delivers output data to other filters after executing a specific data-processing function. The design of such filtering mechanism is often implemented in systems that manipulate large volumes of digital content, such as images or streams of data. Filtering systems should be designed in a way that enables the integration of different types of filters, whether simple, cascade, or composites.

A part of the overall structure of a DCRM system is a filtering subsystem that transforms (manipulates) streams of data. The functionality of this subsystem is achieved by integrating several filters together. The way these filters are integrated and connected controls the format of output. There are several ways of combining these filters. A flexible design structure is required for modeling the complex combination of these filters, which could be hierarchical in nature. The filtering subsystem should provide flexible ways for configuring and integrating filters.

There are several requirements for the design of a filtering subsystem:

- The composition of filters should be robust enough to allow addition of new filters and replacement of existing ones in an easy way.
- The filtering subsystem should provide support to plug in and take out filters without affecting other parts of the system (simplifies maintenance process).
- The filtering subsystem should provide means to select which filters to use dynamically without major code changes (simplifies application customization).
- The filtering subsystem should provide means to integrate filters together and configure the workflow of filtering activities.
- The system should be able to easily arrange the integration and cascading of filters with minimal impact on the system design.
- The filtering subsystem design should not be limited to specific mechanisms for data filtering. It should provide the necessary infrastructure to integrate different types of filters, whether simple, cascade, or composites.

Acquaintance and Retrieval

Since this subsystem is heavily based on filters that process data, we look for design patterns that provide solutions for filtering and data
An obvious choice is to design the filtering subsystem by plugging filters together in a pipeline using the Pipes and Filters pattern [Buschmann et al. 1996]. Pipes and Filters provides a system design as a cascade of processing steps implemented as filters connected by pipes that transfer streams of data between filters. This design gives support for pipes and filters as first-class elements and gives explicit support for sequential calls of filters. However, Pipes and Filters does not support the integration and composition of filters in hierarchical fashion.

The Filter pattern [Grand 1998] can be used to implement sequential calls of filters. It gives explicit support for sources and sinks, and simplifies the sequential combination of filters. However, this design does not support hierarchical combination of filters—filters that are composition of other filters.

We can also consider some general-purpose design patterns. The Composite pattern [Gamma et al. 1995] can be used to implement sequential calls of filters. It gives explicit support for sources and sinks, and simplifies the sequential combination of filters. However, this design does not support hierarchical combination of filters—filters that are composition of other filters.

We can also consider some general-purpose design patterns. The Composite pattern [Gamma et al. 1995] can be used to implement sequential calls of filters. It gives explicit support for sources and sinks, and simplifies the sequential combination of filters. However, this design does not support hierarchical combination of filters—filters that are composition of other filters.

The Filter pattern [Grand 1998] can be used to implement sequential calls of filters. It gives explicit support for sources and sinks, and simplifies the sequential combination of filters. However, this design does not support hierarchical combination of filters—filters that are composition of other filters.

Pattern Selection

In this step we select a set of patterns to use in the design of the filtering subsystem. There are several patterns that we can use to design the subsystem. From the requirements, we must select design patterns that offer hierarchical design of filters. For this purpose, we select the Composite pattern [Gamma et al. 1995] to implement the hierarchical, complex composition of filters. The Composite pattern provides a solution structure for filters that could be composed of other simple filters. The design of the Composite pattern represents part-whole hierarchies of filters. It also provides a mechanism to attach and detach filters.

When the order by which the filters manipulate the data is important, we can use the Pipes and Filters [Buschmann et al. 1996] and the Filter [Grand 1998] patterns. The two patterns have similar designs in which they cascade operation of filters and provide data sources and data sinks. The Pipes and Filters pattern is more architectural in the sense that it treats pipes and filters as first-class design constructs. The Filter pattern assumes that pipes are established as method calls to the next filters. For simplicity, we consider the Filter pattern only in the following discussion. In these particular filters the order in which the filters are cascaded define the filtering solution.

We then want to encapsulate the filtering mechanism, whether cascade, composite, or simple, and provide a unique interface to the rest of the application. We select a Strategy pattern [Gamma et al. 1995] to provide a unique interface to the filtering subsystem. Using this pattern, the filtering strategy is flexible and hidden from any calls and invocations from any other component. This design provides encapsulation of a family of filters, making them interchangeable.

Another feature that we need to support for the filtering subsystem is the flexible creation of the filters and their integration. The AbstractFactory pattern [Gamma et al. 1995] provides an interface for creating families of related objects (filters) without specifying their concrete classes.

POAD Design for the DCRM Filtering Subsystem

Constructing the Pattern-Level diagram
In this step we create instances of the selected patterns and identify the relationships between these instances. As a result, a Pattern-Level diagram of the filtering subsystem is developed.

For the filtering subsystem, we first create pattern instances. In the analysis phase we selected four patterns: Composite, Filter, AbstractFactory, and Strategy. We create a CompositeFilters pattern instance of type Composite to implement the complex hierarchy of filters. We create a CascadeFilters pattern instance of type Filter to implement explicit support for the cascade or pipeline of filter execution. We create a FilterFactory pattern instance of type AbstractFactory to implement the creation and integration of filters, whether simple, cascade, or composite. Finally, we create a FilterStrategy of type Strategy pattern to encapsulate the filtering subsystem.

We then define the dependency relationship between the pattern instances. The FilterFactory pattern instance creates and builds filters of various types. Therefore, the FilterFactory instance has a dependency relationship with the two pattern instances CompositeFilters and CascadeFilters. The FilterStrategy pattern instance provides an interface for the rest of the application to create filtering mechanisms and to invoke the filter pipeline. Therefore, the FilterStrategy pattern instance will have a dependency relationship on the FilterFactory to create the filtering structure and invoke the product created by the factory. Figure 13-10 illustrates the Pattern-Level diagram for the filtering subsystem.

**Figure 13-10. The Pattern-Level diagram for the DCRM filtering subsystem.**

![Pattern-Level diagram for the DCRM filtering subsystem](image)

**Constructing Pattern-Level with Interfaces Diagram**

In this step we analyze the relationships between pattern instances. The dependency relationship between patterns in the Pattern-Level view is a conceptual, high-level uses relationship. We further trace these dependencies to lower-level design.

The FilterFactory pattern instance is of type AbstractFactory pattern, which has two interfaces. The first interface is the AbstractFactory interface class, which provides an interface for creating various types of products (filters). The second interface is for the product created by the factory, the AbstractProduct interface class, which provides an interface for various products (filters) created by the factory.

The CompositeFilters pattern instance is of type Composite pattern, which has one interface: the Component interface class. Component provides a unified interface for a composite structure, whether referring to a simple or hierarchical composition.

The FilterStrategy pattern instance is of type Strategy pattern. Strategy pattern has the Context interface class behind which the filtering policy is encapsulated. Finally, the CascadeFilters pattern instance is of type Filter pattern, which has one interface class, the AbstractSink interface class. The Filter pattern has two different interfaces, depending whether the cascading effect is driven by the source or the sink filter (see Volume 1 of Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML [Grand 1998] for details). For this case study we use one variance, which is driven by the filter source.

We then translate the dependency relationships between pattern instances in Figure 13-10 to pattern interface relationships. The "create and invoke" relationship between FilterStrategy pattern instance of type Strategy and the FilterFactory pattern instance of type AbstractFactory is traced down to dependency relationships between the interface classes offered by the two pattern instances. The Context interface class of the FilterStrategy instance uses the AbstractFactory interface class of the FilterFactory instance to create and establish links between filters. The Context interface class also uses the filtering structure created by the FilterFactory instance, the
AbstractProduct interface class.

The FilterFactory pattern instance creates filters, whether composite or cascades. Therefore, the AbstractProduct interface class of the FilterFactory has a dependency relationship to the Component interface class of the CompositeFilters pattern instance (of type Composite) and the AbstractSink interface class of the CascadeFilters pattern instance (of type Filter).

Figure 13-11 illustrates the Pattern-Level with Interfaces diagram for the filtering subsystem.

Figure 13-11. The Pattern-Level with Interfaces diagram for the DCRM filtering subsystem.

Constructing Detailed Pattern-Level View

To construct the Detailed Pattern-Level diagram for the filtering subsystem, we express the internal structure of each instantiated pattern from the Pattern-Level with Interfaces diagram in Figure 13-11. The structure of the CompositeFilters of type Composite, the FilterStrategy of type Strategy, and the FilterFactory of type AbstractFactory can be found in Design Patterns: Elements of Object-Oriented Software [Gamma et al. 1995].

The structure of the CascadeFilters instance of type Filter can be found in Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML [Grand 1998]. Grand presents two structures for filtering, depending on whether it is a source-driven push mode or a sink-driven pull mode for filtering. We have used a simplified structure. Figure 13-12 illustrates the Detailed Pattern-Level diagram.

Figure 13-12. The Detailed Pattern-Level diagram for the filtering subsystem.
POAD Design Refinement for the DCRM Filtering Subsystem

Instantiating Pattern Internals

In this step we add application-specific nature to the Detailed Pattern-Level diagram by renaming internal pattern classes according to the application design environment, choosing names for pattern participants that are meaningful in the application context, and defining application-specific names for operations in patterns.

Instantiating the internals of the FilterStrategy is illustrated in Figure 13-13. The FilterStrategy pattern instance is composed of

- **FilterContext**. The FilterContext class represents the external environment (the rest of the application) interacting with the filtering subsystem. The FilterInterface() operation is called by other application entities to perform filtering on input data.

- **FilteringStrategy**. The Strategy interface class of the Strategy pattern is renamed FilteringStrategy. It provides the interface for applying filters on data. The mechanism by which filtering is provided is implemented in concrete classes that provide a concrete implementation of the interface operation Apply().

- **ConcreteFilteringStrategy1** and **ConcreteFilteringStrategy2**. These concrete classes provide implementation to the
FilteringStrategy interface class by providing a policy for the filtering data.

**Figure 13-13. Instantiation of the FilterStrategy.**

Instantiating the internals of the FilterFactory is illustrated in **Figure 13-14**. The FilterFactory pattern instance is composed of:

- **AbstractFilterA.** AbstractFilterA plays the role of the ProductA in the AbstractFactory pattern. It defines an interface for a filter of a specific type—for instance, OCR. Similarly, AbstractFilterB defines an interface for another filter type—for instance, an algorithm for title detection.

- **FilteringFactory.** The AbstractFactory class of the AbstractFactory pattern is renamed FilteringFactory. FilteringFactory defines an interface for creating the filter objects for each filter type.

- **ConcreteFilteringFactory1** and **ConcreteFilteringFactory2**. These are concrete implementations of the FilteringFactory interface to attach specific types of filters.

**Figure 13-14. Instantiation of the FilterFactory.**
Instantiating the internals of the CompositeFilters is illustrated in Figure 13-15. The CompositeFilters pattern instance is composed of

- **FilterComponent.** An abstract interface for all filter components. A filter can be a simple filter, a cascaded filter, or a composite filter. FilterComponent defines the interface for applying the filter, `Apply()`, which is the `Operation()` method in the abstract definition of the Composite pattern. Methods are implemented in concrete subclasses according to the type of the filter component.

- **LeafFilter.** LeafFilter is a simple filter that does not call any other filters. By calling this filter, the filtering pipeline terminates in the calling thread. The LeafFilter class implements the `Apply()` method of the abstract class FilterComponent.

- **CompositeFilter.** CompositeFilter represents a hierarchical composition of filters that could be simple or other composite filters. The CompositeFilter class implements the `Apply()` interface to apply all filters in its composition. The CompositeFilter class also implements the methods related to adding a filter to its pool of filters.

Figure 13-15. Instantiation of the CompositeFilters.

---

Instantiating the internals of the CascadeFilters is illustrated in Figure 13-16. The CascadeFilters pattern instance is composed of

- **AbstractFilter.** The AbstractFilter class is the interface for the Filter pattern. It plays the role of the AbstractSink in the design structure of the Filter pattern.

- **SimpleFilter.** The SimpleFilter class is the type of filter that does not do cascading; that is, it is the sink of the cascade operation. It plays the role of the ConcreteSink in the design structure of the Filter pattern.

- **AbstractCascadeFilter.** The AbstractCascadeFilter is an interface for a cascading filter. It plays the role of the AbstractSinkFilter in the design structure of the Filter pattern.

- **ConcreteCascadeFilter.** ConcreteCascadeFilter is a concrete cascade filter that implements the AbstractCascadeFilter
During the creation of a ConcreteCascadeFilter, the next filter to be called is passed as an argument; hence, the client controls the cascading of the filters according to the order required.

Figure 13-16. Instantiation of the CascadeFilters.

Developing an Initial Class Diagram

Starting from the Detailed Pattern-Level diagram in Figure 13-12, we use pattern interfaces and the instantiated details of pattern internals (as obtained in the previous sections) to construct a UML class diagram. This is a simple process of combining the domain-specific details with the Detailed Pattern-Level diagram. The class diagram that is developed at this phase is an initial step to develop the static design model of the pattern-oriented design for the filtering subsystem. In this step we convert relationships between pattern interfaces in the Detailed Pattern-Level to relationships between internal classes of instantiated patterns.

We trace the interface relationship between the Context interface class in the FilterStrategy pattern instance and the AbstractFactory interface class of the FilterFactory pattern instance in Figure 13-12. The Context interface class is renamed FilterContext, as illustrated in Figure 13-13. The AbstractFactory interface class is renamed FilteringFactory, as illustrated in Figure 13-14. Therefore, this pattern interface relationship is translated into an association relationship between the FilterContext class and the FilteringFactory class.

We trace the interface relationship between the Context interface class in the FilterStrategy pattern instance and the AbstractProduct interface class of the FilterFactory pattern instance in Figure 13-12. The Context interface class is renamed FilterContext, as illustrated in Figure 13-13. The AbstractProduct interface class is renamed AbstractFilterA and AbstractFilterB, as illustrated in Figure 13-14. Therefore, this pattern interface relationship is translated into an association relationship between the FilterContext class and the AbstractFilterA and AbstractFilterB classes.
We trace the interface relationship between the AbstractProduct interface class in the FilterFactory pattern instance and the Component interface class of the CompositeFilters pattern instance in Figure 13-12. The AbstractProduct interface class is renamed AbstractFilterA or AbstractFilterB, as illustrated in Figure 13-14. The Component interface class is renamed FilterComponent, as illustrated in Figure 13-15. Therefore, this pattern interface relationship is translated into an association relationship between the AbstractFilterA/B class and the FilterComponent class.

Finally, we trace the interface relationship between the AbstractProduct interface class in the FilterFactory pattern instance and the AbstractSink interface class of the CompositeFilters pattern instance in Figure 13-12. The AbstractProduct interface class is renamed AbstractFilterA or AbstractFilterB, as illustrated in Figure 13-14. The AbstractSink interface class is renamed AbstractFilter as illustrated in Figure 13-16. Therefore, this pattern interface relationship is translated into an association relationship between the AbstractFilterA/B class and the AbstractFilter class.

Figure 13-17 illustrates the initial class diagram for the filtering subsystem.

Figure 13-17. The initial class diagram for the DCRM filtering subsystem.

It can be easily recognized that the patterns are still notable in the class diagram, as shown by the dotted boxes around the classes.

**Design Optimization**

In this step we inspect the initial class diagram for possible optimization, using grouping or reduction. For this subsystem, we used only one pattern instance of each pattern type. Therefore, there is no possible optimization by reducing abstract classes.

We then look at classes with related functionalities. We realize that the products produced by the FilterFactory pattern instance of type AbstractFactory are the filters themselves. The filters could be composite filters, as offered by the CompositeFilters pattern instance, or explicit cascaded filters, as offered by the CascadeFilters pattern instance. Therefore, there is no need to keep the presentation of AbstractFilter of the FilterFactory pattern instance separate from the filters offered through the FilterComponent interface of the
CompositeFilters pattern instance and the AbstractFilter interface of the CascadeFilters pattern instance. Therefore, we eliminate the classes AbstractFilterA and AbstractFilterB and their concrete realizations, and change the dependency of the concrete factories to the FilterComponent and AbstractFilter classes.

Similarly, we change the dependency of the FilterContext from AbstractFilterA to FilterComponent and AbstractFilter classes, as shown in Figure 13-18.

**Figure 13-18. The class diagram of the filtering subsystem after removing the AbstractProducts of the AbstractFactory pattern.**

We now investigate the interfaces for a filter class. The FilterComponent class offers an interface for filters in the CompositeFilters pattern instance. The AbstractFilter class offers an interface for filters in the CascadeFilters pattern instance. The FilteringStrategy class offers an interface for filters as well. All three interfaces provide an interface operation to invoke a filter. The interface for the filtering subsystem can be represented by one class integrating the three interfaces; we call the new class AbstractFilter. AbstractFilter integrates the FilteringStrategy class in the FilterStrategy pattern instance, the FilterComponent class in the CompositeFilters pattern instance, and the AbstractFilter class in the CascadeFilters pattern instance. This interface contains a method to apply the filter, Apply(), which corresponds to the Apply() methods in the FilteringStrategy class of the FilterStrategy pattern instance, the FilterComponent class of the CompositeFilters pattern instance, and the AbstractFilter class of the CascadeFilters pattern instance. It also contains the methods to add and remove components defined in the FilterComponent class of the CompositeFilters pattern instance. Since the three classes are now merged into one class, all concrete classes will implement the new AbstractFilter interface. Some of these concrete classes represent the same concept, a simple filter. Therefore, we merge the ConcreteFilteringStrategy classes of the FilterStrategy pattern instance, the LeafFilter class of the CompositeFilters pattern instance, and the SimpleFilter class of the CascadeFilters pattern instance. We call these classes SimpleFilter in the refined design. Figure 13-19 illustrates the refined class diagram of the Filtering subsystem.

**Figure 13-19. The refined class diagram for the DCRM filtering subsystem.**
Summary

In this chapter, we illustrated the application of the POAD approach to develop the design for two subsystems of a content remastering application: the distribution and the filtering subsystems.

For the distribution subsystem, the use of the patterns provided the necessary transparency to hide the interactions with the remote worker machines from the rest of the application. The combination of the Proxy, AbstractFactory, and Strategy patterns provided flexible means to add new machines to the cluster, communicate with machines running heterogeneous protocols, and encapsulate various load-balancing strategies to distribute tasks across worker machines.

For the filtering subsystem, we used Strategy, AbstractFactory, Composite, and Filter patterns to design the subsystem. The design provides a common interface to the various filtering mechanisms. By using the Strategy pattern, the filtering subsystem provides a common interface to any filter design or any combination of filters that will be used. This design facilitates the integration of the filtering subsystem in the application design. It also minimizes the impact of choosing a different filtering technique or changing the implementation of any filter.

The design supports various filtering mechanisms. By using the Composite pattern, the design of the filtering subsystem supports types of filters that are compositions of simple filters. By using the Filter pattern, the design supports types of filters that consist of sequence of filters. The design of the filtering subsystem is easily configurable. Choosing between various filtering techniques is easy and has no change-impact on the application, since the filtering subsystem provides a common interface. Using dynamic binding, the rest of the application design uses this interface and is not tightly related to the type of filter attached in the filtering subsystem.
Chapter 14. A Medical Informatics System

Medical Informatics Standards

Digital Imaging and Communication in Medicine (DICOM)

Pattern-Oriented Analysis and Design of the DICOM UL

Pattern-Oriented Analysis and Design for the Client Application Entity

Summary
Medical Informatics Standards

Distributed information systems are often too complex to develop from scratch. There is always a great need to reuse existing software components in developing those large systems. The design of such systems involves many frequent design decisions that can be addressed using large-grained design components (e.g., frameworks) and fine-grained design components (e.g., design patterns). For instance, distributed information systems often contain communication subsystems, user interface frameworks, and database solutions. In the process of developing these subsystems, the designer is often challenged by many design problems that can be solved using design patterns. From this viewpoint, the development of such systems is concerned with integrating design components. The POAD process is a component-based development process in which constructional design patterns are used as design components. Hence, POAD is useful in managing the development of large distributed information systems.

Due to their complexity, distributed information systems are often hierarchical in nature. Decomposition of those systems into design patterns is useful in managing the development process. Distributed medical informatics systems belong to such category. In such systems, it is required to provide means to manage the distribution and presentation of medical images within the same medical institution and across institutions. Among the many other functionalities, a medical informatics system manages:

- Storage and retrieval of medical information records and images.
- Transfer of medical information between various medical locations and sites.
- Storage of media.
- Communication protocols between collaborating medical institutions.
- Records of various medical services provided to patients.

Several organizations that develop standards for medical informatics systems are concerned with healthcare-related information in general and medical image exchange in particular. These standards define the messaging system for healthcare data exchange. Medical applications must comply with such standards in order to be able to seamlessly communicate with other medical informatics applications. Connectivity between medical institutions requires sharing complete protocols defined by standards. The Digital Imaging and Communication in Medicine (DICOM) standard [DICOM 2002a; DICOM 2002b] is a standard that defines the communication messages and application services between medical applications.

The American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA) have formed a joint committee to develop DICOM. The DICOM standard is developed according to the NEMA procedures and in liaison with other standard organizations such as ANSI, Health Level Seven (HL7) [HL7 2002], and IEEE. DICOM primary application domain is Information Management Systems (IMS) for medical applications. It defines network protocols, message encoding, object data model, data dictionary, service classes, and conformance requirements. Other standards, such as the HL7, are working to develop interoperability rules that define the manner in which they interact with DICOM systems. Networking is a crucial component for all medical imaging systems, and hence support for open communication standards is a mandate. DICOM builds upon the common well-known communication protocols (OSI, TCP/IP) and defines specification of how application entities, conforming to DICOM, can communicate with each other over these communication protocols.

In this chapter we illustrate the application of the POAD approach to develop a pattern-oriented design for a distributed medical informatics system that implements the DICOM specification. The design that we present here is not meant to provide a unique implementation of the standard, nor does it reflect the opinion of the any of the committees involved in development of the DICOM standard. It is a design that we develop as an illustrative example of how to use the POAD approach to build a large, complex application. In the following section, we elaborate more on the DICOM standard and explain its scope. We illustrate the design of the DICOM Upper Layer (UL) subsystem and the design of a client side application entity (AE) using the POAD process.
Digital Imaging and Communication in Medicine (DICOM)

Specification Versus Implementation

With the increasing need and use of computers in clinical applications, the need emerges for a standardization of image and information transfer between medical devices and institutions. The DICOM standard pertains to the field of medical informatics and facilitates interoperability of medical imaging equipment by specifying:

- A set of protocols to be followed by medical devices.
- The syntax and semantics of commands and associated information.
- Information that must be supplied with an implementation claiming interoperability and conformance to the standard. [DICOM 2002a]

The DICOM standard doesn't provide details regarding any implementations of the standard. It is a specification and not an implementation; hence, it does not define development techniques or methods. It is up to the software vendors to choose the appropriate platforms and technologies to implement the standards. The standard has a conformance guide, which defines the procedures and requirements to verify the compliance of an implementation provided by a vendor to the specifications.

DICOM Scope

The specification is composed of several parts. To establish the necessary background required for the pattern-oriented designs that we illustrate in this chapter, we briefly discuss in this section the technical parts of the DICOM standard. The first two parts introduce the standard and its objectives, and define the conformance requirements of implementations (software programs) seeking conformance to the standard.

Part 3 of DICOM defines information object classes, which provide an abstract definition of the real-world entities applicable to communication of digital medical images. These classes are called Information Object Definitions (IODs). IODs are classified into two types: normalized and composite. Normalized IODs have attributes inherit in the real-world entity. For example, a "Study" conducted on a certain patient is a normalized IOD. The Study IOD has date, time, and other information attributes. The patient's name is not part of the Study attribute because it is inherent in the patient on which the Study was performed. Composite IODs contain attributes that are related to but not inherent in the real-world entity. For example, Computed Tomography Image IOD contains attributes that are inherent in the image (image date) and attributes that are related to but not inherent in the image (patient name). The specification defines IODs for most common medical informatics entities, such as Patient, Study, Visit, Series, Image, and Overlay.

Part 4 defines a number of service classes. A service class associates one or more IODs with one or more commands to be performed on these objects. It specifies requirements for both providers and users of communication services and defines the characteristics shared by all service classes. This part contains a number of normative annexes that describe individual service classes in details. These service classes include Storage Service Class, Query Service Class, Retrieval Service Class, and Study Management Service Class. Service classes are defined in terms of service object pairs (SOPs), and each SOP defines a specific service in the service class. An SOP is a combination of an IOD and the commands that are applicable on that IOD. Commands are defined in terms of messages that can be exchanged between DICOM application entities. Messages are specified in Part 7 of DICOM.

Part 5 specifies how application entities construct and encode the data set information resulting from the use of the IOD and services
classes defined in the parts 3 and 4. It defines encoding rules necessary to construct a data stream to be conveyed in a DICOM message. This data stream is produced from the collection of data elements making up the data set. This part also defines the semantics of a number of generic functions that are common to many IODs.

Part 6 defines the collection of all DICOM data elements available to represent information. This part assigns each data element a unique tag, gives it a name, specifies its value characteristics (character string, integer, etc.), and defines its semantics (i.e., how it is interpreted). Parts 5 and 6 are dedicated to constructing data sets and representing IODs as data sets.

Part 7 specifies the protocols used by an AE in a medical imaging environment to exchange messages over the network. This part defines the command stream of a message. It specifies the rules to establish and terminate associations between application entities, rules that govern the exchange of command requests and responses, and encoding rules necessary to construct command streams and messages. Application entities use and receive a set of message primitives called DICOM Message Service Elements (DIMSE). DIMSE are often grouped in message service groups. These groups are used together with IODs to define SOPs. Messages are exchanged over the network in the form of DICOM messages, which are the encoded version of the DIMSE primitives. A DICOM protocol machine is used to decode and encode DICOM messages and service primitives. The relationship between messages (DIMSE), IODs, SOPs, and service classes is defined in Figure 14-1.

**Figure 14-1. Relationship between DIMSE, IODs, SOPs, and service classes as defined in DICOM standard.**

![Figure 14-1](image)


Part 8 specifies the upper-layer protocols and communication services necessary to support, in a networked environment, communication between DICOM application entities. The communication services specified in this part are a subset of the services offered by the OSI Presentation Service (ISO 8822) and of the OSI Association Control Service Element (ACSE) (ISO 8649). They are referred to as the UL services, which allow peer application entities to establish associations, transfer messages, and terminate associations. The UL service is provided by the DICOM Upper Layer Protocol used in conjunction with TCP/IP transport protocols.

Part 9 is concerned with point-to-point communication between application entities. Parts 10 through 12 are concerned with the media
storage and formats. Part 13 is related to printing services. Parts 14, 15, and 16 are related to display functions, security, and content mapping respectively. Parts 9 through 16 are not considered in the designs developed in this chapter.

The specification is huge and covers lots of aspects related to medical imaging informatics. The design we illustrate in this chapter benefits mainly from the parts of the DICOM standard illustrated in Figure 14-2 taken from the DICOM standard.

Figure 14-2. Parts of the DICOM standard.

A Client/Server Architecture

A system that implements all of the DICOM specification is large and complex. Usually, vendors implement portions of the specifications. Implementations usually cover a subcategory of services, or services that are related to a particular medical device.

The first analysis step is to decompose the system at a high level into large components (subsystems) and define their interconnections. Figure 14-3 illustrates a high-level structure of the medical informatics system as a client/server application connected via a network. The DICOM specifies the transport and presentation layers for a TCP/IP network protocol such as DICOM UL (DICOM UL client and DICOM UL server subsystems). At a very high level the client side is composed of the two subsystems: the application entity as a client (Client AE) and the DICOM UL client. The Client AE plays the role of the service class user (SCU) as specified in DICOM standard [DICOM 2002a]. The server side is composed of the application entity as a server (Server AE) and the DICOM UL server. The Server AE plays the role of a
Figure 14-3. A high-level decomposition of a client/server medical informatics application.
Pattern-Oriented Analysis and Design of the DICOM UL

Networking is a crucial component for all medical imaging systems, and hence support for open communication standards is a mandate. The communication protocols specified in DICOM standards are closely related to the ISO Open System Interconnection Basic Reference Model [ISO 1997]. The UL services allow peer application entities to establish association and transfer messages. There are three options for communication in DICOM:

- A minimum OSI stack of protocols with a full-duplex Session Kernel, Presentation Kernel, an ACSE,
- A UL protocol augmenting TCP/IP, or
- A point-to-point protocol stack.

In the following discussion, we illustrate the design for one DICOM communication solution; the DICOM UL protocol augmenting TCP/IP, because it combines the OSI UL protocols into a simple-to-implement, single protocol while providing the same services offered by the OSI stack.

POAD Analysis for DICOM Upper-Layer Subsystem

Requirements Analysis

For this case study, the DICOM specification documents define the functional requirements for the communication subsystem. We start by reviewing the terminologies used in the specification related to the communication subsystem functionalities:

- **DICOM UL**: The UL protocol is related to the session, presentation, and part of the application layer of the ISO reference model.
- **DIMSE**: The DICOM Message Service Element is a set of DICOM application-layer communication messages. These messages define the possible services, such as storing and retrieving images and queries.
- **PDU Messages**: Protocol Data Unit (PDU) messages are the messages exchanged between peer entities within a layer in the ISO reference model. A PDU consists of protocol control information and user data.
- **Message Primitives**: The messages exchanged between the application entities and the UL services entities (DICOM UL).

The following abbreviations are not part of the DICOM standard, but we introduce them to facilitate the readability of diagrams: PRIM for primitive message, RP for response message, IND for indication message, RQ for request message, CNF for confirmation message, and, as mentioned, AE for application entity.

The network communication services, specified in DICOM, constitute a set of generic services provided to support the communication of DICOM AEs. They are referred to as the Upper-Layer service, or UL service. This definition of the UL service allows the use of a fully conformant stack of OSI protocols to achieve robust and efficient communication. In this section we are concerned with the
communication of medical application entities over TCP/IP, which is commonly used and is the Internet communication standard.

Figure 14-4 describes a three-layer architecture for a medical AE communicating with another entity, using TCP/IP as their lower communication protocol and DICOM message primitives as their application communication protocol. Both the client and server sides have the same high-level structure.

**Figure 14-4. The high level architecture of a DICOM client/server.**

The DICOM UL communicates with the DICOM AE using application messages, PRIMs. The DICOM PRIMs are the means of communication between DICOM application entities distributed over the network; that is, a DICOM AE can understand only DICOM PRIMs regardless of the format that was used to transfer the message over the network. The DICOM UL formulates PDU messages to be sent to the TCP/IP stack. PDU messages encapsulate the DICOM PRIMs to be exchanged. A DICOM PRIM can be one of the following types: request primitive (RQ PRIM), indication primitive (IND PRIM), response primitive (RP PRIM), and confirmation primitive (CNF PRIM). Figure 14-5 illustrates an example of message exchange between two application entities. The client AE initiates a request (RQ PRIM), which will be received by the server AE as an indication message (IND PRIM). The server replies with a response message (RP PRIM), which will be received by the client as a confirmation (CNF PRIM).

**Figure 14-5. Message primitives.**
For each message exchange between DICOM AEs, the four PRIMs, shown in Figure 14-5, are used to transfer peer-to-peer application messages. For example, the following are some application message types that are carried by the message primitives:

**A_ASSOCIATE**: Establishment of an association between two AEs is performed by using the request, indication, response, and confirmation primitives for an application message called A_ASSOCIATE.

**A_RELEASE**: An AE that desires to release the association with another AE will use the application message A_RELEASE, referred to as REL.

**A_ABORT**: Used by a requestor AE to cause the abnormal release of the association, referred to as "ABORT"

**A_P_ABORT**: Used by the UL DICOM to signal the abnormal release of the association due to problems in the services at the presentation layer and below (mainly the communication loss between two entities).

**P_DATA**: Used by an AE to cause the exchange of application information.

*Figure 14-6* illustrates an example of a communication scenario between a client and a server for initiating communication and establishing the link. When a DICOM UL becomes activated (Association Idle State), it waits for TCP transport connections in a passive mode by initiating a "listen." When an association is to be established by an AE, an ASC-RQ primitive is passed from the AE to the UL, which issues a transport connect (TC) request primitive to the TCP Transport Service (Active Open). Once the TCP Transport Connection Confirmation is received (Open Completed), an A-ASSOCIATE-RQ PDU is sent/written on the now established transport connection. When an incoming TCP Transport Connection Indication is received from the network (Server Side), it is accepted, and a timer ARTIM (Association Request/Reject/Release Timer) is set.

*Figure 14-6. A collaboration between client/server entities for association establishment scenario.*
It is clear from the above discussion that the DICOM UL is a sophisticated layer. It handles many messages and establishes communication between AEs on different machines such that the communication between the entities is made transparent regardless of the underlying communication protocol and communication details. The DICOM UL plays a very important role in handling messages, establishing communication links, and transferring information to and from remote entities.

When studying the specification of this layer in the DICOM standard, we find that it is a heavily state-based engine that reacts to many events. This layer has many states, and depending on its state, the behavior of the layer is determined. The standard specifies the DICOM UL behavior over TCP/IP, using a state table [DICOM 2002a]. To understand the complexity of the layer, Table 14-1 lists the states and the actions that the layer exhibits.
<table>
<thead>
<tr>
<th>States</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>Idle</td>
</tr>
<tr>
<td>State 2</td>
<td>Transport connection open (awaiting ASC_RQ PDU)</td>
</tr>
<tr>
<td>State 3</td>
<td>Awaiting local ASC_RP PRIM (from local user)</td>
</tr>
<tr>
<td>State 4</td>
<td>Awaiting transport connection opening to complete (from local transport service)</td>
</tr>
<tr>
<td>State 5</td>
<td>Awaiting ASC_AC or ASC_RJ PDU</td>
</tr>
<tr>
<td>State 6</td>
<td>Association established and ready for data transfer</td>
</tr>
<tr>
<td>State 7</td>
<td>Awaiting REL_RP PDU</td>
</tr>
<tr>
<td>State 8</td>
<td>Awaiting local REL_RP PRIM (from local user)</td>
</tr>
<tr>
<td>State 9</td>
<td>Release collision requestor side; awaiting REL_RP PRIM (from local user)</td>
</tr>
<tr>
<td>State 10</td>
<td>Release collision acceptor side; awaiting REL_RP PDU</td>
</tr>
<tr>
<td>State 11</td>
<td>Release collision acceptor side; awaiting REL_RP PDU (from local user)</td>
</tr>
<tr>
<td>State 12</td>
<td>Awaiting transport connection Close Indication (Association no longer exists)</td>
</tr>
</tbody>
</table>

**Actions Related to Association Establishment**

AE-1  Issue TRANSPORT CONNECT request primitive to local transport service; next state is State 4

AE-2  Send ASC_RQ PDU; next state is State 5

AE-3  Issue ASC_CNF PRIM; next state is State 6

AE-4  Issue ASC_CNF (reject) PRIM and close transport connection; next state is State 1

AE-5  Issue transport connection response primitive; start ARTIM timer; next state is State 2

AE-6  Stop ARTIM timer, and if ASC_RQ is acceptable by service-provider, issue ASC_IND PRIM; next state is State 3; otherwise, issue ASC_RJ PDU and start ARTIM timer; next state is State 13

AE-7  Send ASC_AC PDU; next state is State 6

AE-8  Send ASC_RJ PDU and start ARTIM timer; next state is State 13

**Actions Related to Data Transfer**

DT-1  Send DATA-TF PDU; next state is State 6

DT-2  Send DATA IND PRIM; next state is State 6

**Actions Related to Association Release**

AR-1  Send REL_RQ PDU; next state is State 7

AR-2  Issue REL_IND PRIM; next state is State 8
### States and Actions

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR-3</td>
<td>Issue REL_CNF PRIM and close transport connection; next state is State 1</td>
</tr>
<tr>
<td>AR-4</td>
<td>Issue REL_RP PDU and start ARTIM timer; next state is State 13</td>
</tr>
<tr>
<td>AR-5</td>
<td>Stop ARTIM timer; next state is State 1</td>
</tr>
<tr>
<td>AR-6</td>
<td>Issue DATA IND; next state is State 7</td>
</tr>
<tr>
<td>AR-7</td>
<td>Issue DATA-TF PDU; next state is State 8</td>
</tr>
<tr>
<td>AR-8</td>
<td>Issue REL_IND (release collision): If association-requestor, next state is State 9; otherwise, next state is State 10</td>
</tr>
<tr>
<td>AR-9</td>
<td>Send REL_RP PDU; next state is State 11</td>
</tr>
<tr>
<td>AR-10</td>
<td>Issue REL_CNF primitive; next state is State 12</td>
</tr>
</tbody>
</table>

### Actions Related to Aborting the Session

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA-1</td>
<td>Send ABORT PDU (service-user source) and start (or restart if already started) ARTIM timer; next state is State 13</td>
</tr>
<tr>
<td>AA-2</td>
<td>Stop ARTIM timer if running; close transport connection; next state is State 1</td>
</tr>
<tr>
<td>AA-3</td>
<td>If service-user initiated abort, issue ABORT IND and close transport connection; if service-provider initiated abort, issue P-ABORT IND and close transport connection; next state is State 1</td>
</tr>
<tr>
<td>AA-4</td>
<td>Issue P-ABORT IND PRIM; next state is State 1</td>
</tr>
<tr>
<td>AA-5</td>
<td>Stop ARTIM timer; next state is State 1</td>
</tr>
<tr>
<td>AA-6</td>
<td>Ignore PDU; next state is State 13</td>
</tr>
<tr>
<td>AA-7</td>
<td>Send ABORT PDU; next state is State 13</td>
</tr>
<tr>
<td>AA-8</td>
<td>Send ABORT PDU (service-provider source-), issue a P-ABORT_IND, and start ARTIM timer; next state is State 13</td>
</tr>
</tbody>
</table>

---

**Figure 14-7** is the state table of the DICOM UL as specified in DICOM Part 8^DICOM 2002a, DICOM 2002b^.

**Figure 14-7. The state table of DICOM UL.**

Acquaintance and Retrieval

Based on the analysis of the specification that we conducted in the previous step, we conclude that the DICOM UL is a state-based component with complex behavior. Therefore, we search for patterns that we can use to design components with complex state-based behavior, which is usually the case for reactive systems.

In exploring several patterns repository, we retrieve a set of patterns and pattern languages that are useful in implementing a state-based component with complex behavior. The selected patterns include the following candidates. The State pattern [Gamma et al. 1995] defines how to design an object whose behavior changes according to its states. This pattern is also referred to as State Object [Dyson & Anderson 1998]. Robert Martin (1995) also has a three-layer Finite State Machine (FSM) pattern. Dyson and Anderson (1998) present a set of patterns that provides solutions to several design problems that we encounter in designing a finite state machine for an object with complex behavior. The patterns in that collection include State Object, Owner-Driven Transitions, and Exposed State.

A comprehensive collection of design patterns for state-based objects is discussed in "Finite State Machine Patterns" (1999i), a work by the authors of this book. This set of patterns constitutes a pattern language of FSMs that addresses design issues related to the machine type (Meally, Moore, or hybrid), the design structure (layered or interface organization), exposure of an entity's internal state (exposed or encapsulated state), and the instantiation technique of state objects (static or dynamic state instantiation).

Recall from the discussion in the previous section that the behavior of the DICOM UL is complex. Therefore, the structure of the state machine will be complex as well. To manage the complexity of state machines, statecharts can be used. Statecharts extend FSMs with new concepts such as hierarchy, orthogonality, broadcasting, and history states. The hierarchy principle introduces a more global state, referred to as a superstate, which includes other states. The principle of statecharts that allows several behaviors to be experienced at the same time is called orthogonality. The broadcasting principle allows events produced in one state to be broadcasted to orthogonal states.

A pattern language for statecharts is presented in "A Pattern Language of Statechart" (1998b), a work by the authors. The pattern language offers solutions to frequent design problems that commonly confront a system designer in designing statecharts. These problems include how to implement specifications containing hierarchy, orthogonality, and broadcasting. The pattern language maps the specification into object-oriented design and hence code can be easily generated. A set of five statechart patterns—Basic Statechart, Hierarchical Statechart, Orthogonal Behavior, Broadcasting, and History State—are documented in that collection.
Pattern Selection

Since the DICOM UL is a state-based subsystem, we use state machine patterns. Since the behavior of the UL is complex, it is better to manage the complexity using statecharts. Thus, we use the Statechart patterns [Yacoub & Ammar 1998b]. Those patterns build on the design of the FSM pattern language in [Yacoub & Ammar 1999].

The specification does not define the FSM or the statechart of the DICOM UL. Instead, it uses a state table (see Figure 14-7). The state table representation has the following disadvantages:

- From the specification perspective, it is difficult to understand because the tabular form is harder to comprehend than graphical forms such as state machines and statecharts.
- From the implementation perspective, an application developer finds the system hard to understand because when two medical applications are in conversation, one plays the role of a client while the other is the server. The state table specification does not distinguish the role played by each in the state table. Conformance to a specific role would be easier to achieve if the client and the server roles are separately specified.
- The state table, like the state diagram, is flat in nature. They do not possess a hierarchical description of the systems; thus, they tend to be complicated and cumbersome for large, complex systems.

As a result, we translate the state table into statechart specification, which will make it easier for us to use statechart patterns to design the DICOM UL. This process represents a feedback loop in the POAD process where we find at the pattern selection step that we need to go back to the requirement analysis phase to translate the state table to a statechart. It also represents an example of how the existence of some patterns in the database could influence the analysis and design process. For instance, if we do not have statechart patterns in the database, we could proceed with a design that uses state tables and traditional object-oriented design.

Initially, we started with a state diagram description, but this led to a highly sophisticated and complex specification. The use of the hierarchy principle of statecharts simplified the diagram and specification of the system. We also separate the client and the server specification. An application claiming conformance to the DICOM standard can be a client, a server, or both. We first specify the statechart of the server; similarly, that of the client is specified. Figures 14-8 and 14-9 describe the statecharts of the server and client respectively.

Figure 14-8. The statechart specification of DICOM UL as a server.
Figure 14-9. The statechart specification of DICOM UL as a client.
POAD Design for DICOM Upper-Layer Subsystem

Constructing Pattern-Level Diagrams

In this step, we create instances of the selected patterns and identify the relationships between these instances. As a result, a pattern-level logical view of the system is developed. However, as we have noticed, the whole DICOM UL subsystem can be designed using FSM and statechart patterns. Figure 14-10 illustrates the set of design patterns for FSMs. We summarize the design problem solved by each pattern in the following; for further details, refer to “Finite State Machine Patterns” [Yacoub & Ammar 1999].

Figure 14-10. Finite state machine patterns.
The basic FSM pattern (Basic FSM) is an extension of the State pattern [Gamma et al. 1995, also referred to as State Object Dyson & Anderson 1998]. It adds implementation of the state transition diagram specifications, such as actions, events, and a state transition mechanism. The Basic FSM is classified, according to the state transition mechanism, as Owner-Driven Transitions and State-Driven Transitions, which are in tension with each other.

For maintainability purposes, the Basic FSM design can be structured into layered organization and interface organization. The layered organization splits the behavior and the logic transitions so that the machine can be easily maintained and comprehended. The interface organization allows the design to be embedded into the overall application design and facilitates communication between other entities and the machine design.

According to the mechanism for producing an FSM output (i.e., the machine type [Roth 1975]), the Basic FSM is extended into Meally, Moore, or hybrid to describe whether the outputs are dependent only on the entity’s current state or on the events as well.

The entity described by an FSM has a particular state at a given time. The current state of the entity can be exposed to other AEs to allow direct invocation of the state class methods (i.e., Exposed State). It also can be encapsulated inside the entity, and no access is permitted from other AEs (i.e., Encapsulated State). The designer uses one of these two patterns by choosing either to expose the entity’s current state or prevent access to it.

Figure 14-11 illustrates the set of design patterns for statecharts. We summarize the design problem solved by each pattern in the following; for further details we refer to “A Pattern Language of Statechart” [Yacoub & Ammar 1998b].

Figure 14-11. Statechart patterns.
The Basic Statechart translates the elements of statechart specifications into an object-oriented design. Based on the statechart's hierarchy principal, the Hierarchical Statechart extends the basic pattern to support hierarchical states in which a superstate is composed of other states. Sometimes the entity's behavior is described using orthogonal, non-contradicting behaviors by means of the statechart's AND-decomposition specification; thus the Hierarchical Statechart is extended to Orthogonal Behavior to support handling events to orthogonal states. Using an Orthogonal Behavior may lead to the possibility of broadcasting the effect of an event in a state to another orthogonal state, Broadcasting extends Orthogonal Behavior to support event broadcasting. Finally, the statechart's history specification is sometimes used in the superstates of the Hierarchical pattern; History State addresses this problem.

Now we want to determine which pattern of these pattern collections we should use and how they interface to each other. In examining the internal design of each of those patterns, we find that they are all built on the same structure. Though each pattern solves a different design problem, the solution is usually an addition to the internal design elements. Thus, in reality we are using one design structure and, based on the design issues, we alter the internal details.

At the Pattern-Level diagram we are only interested in the patterns we use and how they interface to each other. Therefore, we will use only one pattern instance to design the DICOM UL. We will call it the DICOM-UL pattern instance of type Statechart pattern, as shown in Figure 14-12.

**Figure 14-12. The Pattern-Level diagram for DICOM UL.**

Constructing Pattern-Level with Interfaces Diagrams

In this step we declare the interface for the pattern instances and analyze the relationships between interfaces. Since there is only one pattern instance, we represent the pattern instance and its interface `Entity_Interface` class, which is the interface of the Statechart pattern. See Figure 14-13.

**Figure 14-13. The Pattern-Level with Interface diagram for DICOM UL.**
Constructing Detailed Pattern-Level Diagrams

To construct the Detailed Pattern-Level diagram, we express the internal parts of each instantiated pattern from the Pattern-Level with Interfaces diagram. The internal design structure for the Statechart pattern can be found in "A Pattern Language of Statechart" [Yacoub & Ammar 1998b]. Figure 14-14 illustrates the Detailed Pattern-Level diagram.

Figure 14-14. The Detailed Pattern-Level diagram.

POAD Design Refinement for the DICOM Upper-Layer Subsystem
Instantiating Pattern Internals

In this step we add application-specific nature to the Detailed Pattern-Level diagram by renaming internal pattern classes according to the application design environment, choosing names for pattern participants that are meaningful in the application context, and defining application specific names for operations in patterns.

We instantiate only one pattern instance: the DICOM UL instance in Figure 14-14. In "A Pattern Language of Statechart" (Yacoub & Ammar 1998), we define a pattern language for designing statecharts. The patterns provide solutions to implement hierarchy, orthogonality, and broadcasting in a statechart's object-oriented design. Based on the structure of these patterns, the UML class diagram of the DICOM UL can be derived. We derived the statechart specification for the client side DICOM UL (Figure 14-9) and the server side DICOM UL (Figure 14-8) using the state table specification in the DICOM standard (Figure 14-7). In the following we illustrate the instantiation of the server side DICOM UL (just because it has more diversity in types of states); the client side can be similarly produced.

From Figure 14-9, we find that the statechart is hierarchical. We can identify simple states that do not belong to any state hierarchy, such as Idle, Await_ASC_RQ_PDU, and Await_TC_Closed_IND. The ActiveSession is a top superstate that has intermediate and leaf states. The AssociationRelease state is an intermediate state that is a child of the ActiveSession state and contains other leaf states, such as Await_REL_RP_PDU and Await_REL_RP_PRIM. Therefore, from Figure 14-9, we are able to distinguish simple, leaf, intermediate, and top states. To instantiate the statechart pattern, we create an internal state class that inherits from either the SimpleState, LeafState, IntermediateState, or TopSuperState classes depending on the type of the states.

We then instantiate the Events class. In the Events class we define an abstract method for each event in the statechart. Each concrete state then implements the event based on how the object should respond to the event in a given state. By inspecting the statechart specification in Figure 14-8, we can identify the events that occur in the statechart by tracing the transition arrows from one state to the other. For example, we can identify the ASC_RQ_PDU between the Transport_Connection_Open state and the ActiveSession state, the ASC_RP_PRIM_Accept between the Await_Local_ASS_RP state and the DataTransfer state.

We then instantiate the Actions class. In the Actions class, we define and provide a default implementation for each action that the DICOM UL can execute. This class represents the repository of actions that the DICOM UL can execute from any state. According to the requirement analysis phase, we have identified and grouped a set of actions that the DICOM UL can execute—for instance, the action AE5 (issue transport connection response primitive) and the AA1 (send ABORT_PDU).

We then instantiate the Entity_Interface class. We rename the class to DICOM_Interface rather than the Entity_Interface. Hence, the internal reference to the DICOM_Interface in the AState class will also change.

Figure 14-15 illustrates the instantiated class diagram for the DICOM UL server. A similar diagram could be instantiated for the client side DICOM UL. For simplicity, we include only a subset of the possible actions and events in the figure.

Figure 14-15. The UML class diagram for DICOM UL statechart as a server.
We find the statechart specification (that we developed in the analysis phase) beneficial over the state table defined in DICOM standard because

- We are able to use a design component—that is, the statechart pattern—to implement the behavior of the DICOM UL subsystem.
- The clients and the server behaviors are identified separately; hence, an implementation can claim conformance to one or both.
- The use of pattern language of statechart directly maps the specification into maintainable OO design.
- Statechart is a specification language, and hence we can further verify and validate the correctness of the state table specified in the standard.

### Developing an Initial Class Diagram

Since there is only one pattern structure used to design the DICOM UL, the initial class diagram for the subsystem will be the class diagram for the instantiated pattern, as shown in Figure 14-15.

### Design Optimization

Only one pattern instance is used in designing the DICOM UL subsystem; hence, there is no opportunity to perform any merging or grouping activities in the initial class diagram.

### POAD is Simple Process for Designing Reactive Systems

By observing the POAD steps in the above sections, we notice that the POAD process is very simple to apply. For those types of systems where the architecture is a composition of a set of components and each component has state-based complex behavior, the POAD process is easy to use. This is because the specification of components with complex behavior can be captured using statecharts or FSMs. For those specifications, a set of pattern languages for FSM [Yacoub & Ammar 1999] and statecharts [Yacoub & Ammar 1998b] can be used to design each component. Hence, the analysis phase of POAD is simplified because the analyst knows of the existence of a pattern language that can be used to design each component. The design-refinement phase is also simplified because there is no need for optimization through reduction or grouping. The implementation phase is also simplified, since the FSM and statechart pattern languages
map the specification into object-oriented design, and hence code can be easily generated.

Many systems are developed as concurrent executing components; each component has a complex behavior that is specified using state machines and statechart. The Real-time Object Oriented Modeling [ROOM 1994] is one software development approach that is based on this concept. FSM and statechart patterns can be used as design components in building these systems entirely with patterns. Hence, the specification models are directly mapped to OO designs using our POAD process and the FSM and statechart patterns.
Pattern-Oriented Analysis and Design for the Client Application Entity

In this section we develop a pattern-oriented design for the client side AE. We limit our discussion to the high-level structure without delving into the details of the capabilities and DICOM services that a client side AE could support. We also include the DICOM UL as a design component with which the client AE interacts, as previously illustrated in Figure 14-4.

POAD Analysis for the Client Application Entity

Requirements Analysis

The client side of a distributed medical informatics system, as inferred from the DICOM standard, is composed of the Client AE and DICOM UL client subsystems. Client AE plays the role of the SCU, and the DICOM UL Client provides the TCP/IP communication services and OSI presentation and ACSE services. The DICOM UL is discussed in the previous section and is used here as a design component implemented using statechart patterns. We focus in the following discussion on the Client AE subsystem.

The Client AE subsystem provides the medical informatics system user with the interface that handles services initiated by the user as a client. Therefore, it should possess a user interface. Client requests are mapped into commands that correspond to specific services. Examples of the command initiated by the user include retrieval of an image, storage of an image, storage of a study conducted on a patient, retrieval of the information about a specific visit, and retrieval of information about a patient.

The real-world activities in a DICOM application are represented by one or more computer information metaphors, the SOP components. The functionalities requested by the DICOM system user (i.e., the actual application client) should then be internally expressed as SOPs. A command interpreter is required to interpret the client requests and create the appropriate SOP component. In this way, we decouple the user interface and the design of the user interface commands from the internal DICOM representation of services and information objects.

An SOP component is the union of a specific set of messages used to request some function and one related IOD. The combination of the function requests in terms of messages and the information related to the request defines a precise context for communication between the requester (client) and the server (providing the SOP service). Messages exchanged between DICOM application entities and internally within a DICOM application have two formats. The first format is DIMSE. These messages are generated and received by the SCU. The SCU implements the request for a specific service using an SOP class. DIMSE messages are defined in Part 7 of the DICOM standard, and they define the kind of service (or function) to be performed. These messages include, for example, C_GET to get some information, C_MOVE to move information from one location to another, C_FIND to search for specific information, and N_SET to set a part of the information as defined in the IOD. DIMSE messages carry information about the values of attributes of the IOD to be processed with the message. DIMSEs are also classified according to the nature of the IOD: composite or normalized, as discussed earlier. DIMSE messages are grouped with an IOD to perform a certain service.

To send a service request, the DIMSE message (which now carries the request according to the message type and the information for some IOD) must be sent to the server entity. Messages are not transferred to other DICOM applications using the DIMSE format and structure. They are encoded according to rules and restrictions specified in Part 7 of the DICOM standard. The encoding phase mainly builds what is called a DICOM message (a second message format) from the DIMSE messages. A protocol machine should be responsible for building DICOM messages as command sets and data sets using the DIMSE messages.

According to the standard, not every DIMSE message generated by an SCU should be translated into DICOM messages. For example, the association establishment messages are not encoded into DICOM messages. Therefore, some messages generated by the SCU need to be encoded, and others (association messages) are sent directly to the DICOM UL. Similarly, messages received from the DICOM UL are either passed directly to the SCU (association messages) or are decoded by the protocol machine. A message coordinator...
component is required to handle messages to and from the DICOM application and determine the route of the message according to its type.

As a conclusion of this short analysis, we determine that in order to define the architecture of the client side, we need the following conceptual components: Client User Interface, Command Interpreter, Service Object Pairs, Protocol Machine, Message Coordinator, and DICOM UL Client.

**Acquaintance and Retrieval**

Based on the analysis of the previous section, we find that we are addressing a variety of design problems and issues, which we should be searching for patterns to solve. In the following we illustrate samples of the searches for such solutions.

First, the client side application should have a graphical user interface (GUI) to make it easy for the client to interact with the client DICOM AE. We look for patterns that are commonly used in the design of GUIs. In browsing a collection of available patterns for GUI design, we classify the GUI patterns as (a) patterns related to the interface with the user, including structure and logical layout; and (b) architecture patterns that define how the user actions are translated into actions in the application at the backend. As an example of the first category, the analyst might consider the display maintenance collection of patterns [Towell 1999] or the pattern language for developing form-style windows [Bradac & Fletcher 1998]. As an example of the second category, the analyst might consider the Model View Controller (MVC) pattern [Johnson 1995; Buschmann et al. 1996] or the Presentation-Abstraction-Control pattern [Buschmann et al. 1996]. Those patterns are just few examples of many design patterns for the user interface design.

As part of the search for patterns, according to the outcome of the requirements analysis step, we need to look for patterns to communicate user requests to other application entities that handle the requests or commands. Command design solution can be found (for example) in the Command pattern [Gamma et al. 1995] and the Command Processor pattern [Sommerlad 1996; Buschmann et al. 1996]. The Pattern Almanac [Rising 2000, pp xxxiv] contains a list of patterns under the GUI development category.

Every service is associated with the creation of an SOP. The creation process of those objects needs to be handled by a builder or a factory that interprets the commands and creates the appropriate service and information objects. Several design patterns can serve that purpose: Abstract Factory, Factory Method, and Prototype, for example, which are mostly general-purpose patterns [Gamma et al. 1995].

**Pattern Selection**

Using the conceptual components identified in the requirements analysis and the set of candidate patterns (some of which are discussed in the previous section), we select patterns that best satisfy the responsibilities allocated to each component.

**The Client User Interface**

The MVC [Krasner & Pope 1988; Johnson 1995] is often used to build user interfaces. MVC consists of three components: the model that is related to the application itself, the view that is the user interface representation, and the controller that defines how the user interface reacts to user input. MVC is used to decouple the functionalities of user interfaces and to increase flexibility and reuse. There are several user interface frameworks. The MVC is just one example that we select for this case study.

**Command Interpreter**

The client MVC pattern identifies the command requested by the user and updates the view accordingly. This command is interpreted by a
command interpreter to identify which functionality and service class is required. The command interpreter encapsulates the request as an object. Each request object is then responsible for the creation of the requested SCU. The Command pattern \[\textit{Gamma et al. 1995}\] is selected for this function. This pattern decouples the object that invokes the operation from the one that knows how to perform it. It also provides the flexibility to extend the design with other requests because commands are used as first-class objects. Using the command pattern, it becomes easier to add new commands because changes to existing classes are minimized.

**Service Object Pairs**

The creation of an SOP is not a simple procedure. Depending on the size of the client and the possible requests that it can initiate, the client AE may support several SOPs and service classes. The design of the client should be flexible enough to support future incorporation of other SOPs as the client claims conformance to these SOPs. The creation of SOPs should be provided by an interface for creating families of related IOD and DIMSE. An Abstract Factory pattern is used for this design problem. Each concrete factory represents an SOP. The concrete factory is responsible for creating products, which are IODs and DIMSE messages. These concrete factories play the role of the SCU in the application design. SCU is the role played by the DICOM AE that invokes operations and performs notifications on a specific association. Examples of SOPs are CT Image as an IOD and C_STORE messages, Study IOD and C_Find messages, and Patient IOD and C_Get messages.

**Protocol Machine**

The protocol machine is responsible for building DICOM messages from DIMSE constructs. For instance, the C-STORE request and indication DIMSE-C message are converted to the DICOM C-STORE-RQ message. It is also responsible for decoding DICOM messages into DIMSE messages. For instance, the information necessary for the C-STORE response and confirmation DIMSE-C message is converted into a DICOM C-STORE-RP message. The construction process is merely an encoding process, which defines the command and the data fields of the DICOM messages as specified in the DICOM standard. The encoding of the data set field is defined in Part 5 of the standard, and the command field is defined in Part 7. The construction process is complex. A Builder pattern is used to facilitate the process of constructing complex messages using its constituent parts.

**Message Coordinator**

To coordinate message exchange between the DICOM UL, the SCU (implementing an SOP), and the protocol machine, a message coordinator is used. Message coordinator is implemented by the Mediator pattern. The mediator defines an object that encapsulates how sets of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly and hence provides flexibility in handling and routing messages between objects. This feature is very useful in the design of the Client AE, since some of the DIMSE messages will be encoded or not based on its type, and hence we have to direct the message to different components accordingly.

**DICOM UL Client**

The DICOM UL provides the OSI service related to presentation and session services. It also provides services related to association establishment. The DICOM UL has complex behavior that is specified by a statechart, and a Statechart pattern is used to design and implement this complex behavior.

As a product of the POAD analysis phase, we selected the following patterns for the client AE design: an MVC pattern for the GUI, a Command pattern to encapsulate various client requests in a flexible form, an Abstract Factory pattern to create SOPs, a Builder pattern to build DICOM messages, a Mediator pattern to organize message routing among objects, and a Statechart pattern for the communication subsystem.
POAD Design for the Client Application Entity

Constructing Pattern-Level Diagrams

In this step we create instances of the selected patterns and identify the relationships between these instances. As a result, a Pattern-Level diagram of the Client AE system is developed. See Figure 14-16

Figure 14-16. A Pattern-Level diagram for client side of a medical informatics application.

First, we create pattern instances. A ClientUserInterface of type MVC pattern is used as an example to model the user interface. The specific implementation of an MVC is not discussed here, since it is dependent on the specific nature of the DICOM user interface that needs to be created and details about what should exist in that view. As a design component, we are interested in the Model component of the MVC pattern, since it will be glued by other patterns in the DICOM application.

A CommandInterpreter of type Command pattern is used to interpret the user interface commands and create or delegate the request to the appropriate service objects. A DICOM service is modeled by an SOP of type Abstract Factory pattern. Each SOP is a composition of DIMSE and one or more IODs. A ProtocolMachine of type Builder pattern is used to encode and decode messages and primitives. A
Message Coordinator of type Mediator pattern is used to manage the exchange of messages between components. Finally, the DICOM UL of type Statechart pattern is used to model the UL services of DICOM.

Second, we define dependencies between patterns. The ClientUserInterface pattern instance sends the user commands to the CommandInterpreter instance, which accordingly creates and delegates the processing of the request to the SOP factory instance. The SOP creates the DIMSE messages and the IODs necessary for processing the command. The DIMSE message is sent to the MessageCoordinator instance, which determines whether or not to decode the DIMSE into a DICOM message using the ProtocolMachine instance. The MessageCoordinator then sends the DIMSE or the DICOM message as a request primitive to the DICOM-UL instance. Similarly messages from the DICOM-UL instance received from the server AE across the network are routed to the SOP and then to the client.

**Constructing Pattern-Level with Interfaces Diagrams**

In this step we analyze the relationships between pattern instances. The dependency relationship between patterns in the pattern-level view is a conceptual, high-level uses relationship. We further trace these dependencies to lower-level design.

First, we declare interfaces for the patterns used in the Pattern-Level diagram. For the MVC pattern, we use the Model class as an interface. The Command pattern has the Invoker and Receiver as interface classes. The Abstract Factory class is the interface for the Abstract Factory pattern. The Builder pattern has the Director as an interface class that receives messages and primitives to build the corresponding encoded messages. The Statechart pattern has the class Entity_Interface as its interface. The Mediator pattern has interface as a set of Colleagues for which the mediation of messages is required.

Then, we identify the relationship between pattern interfaces by translating all dependency relationships between patterns in a Pattern-Level diagram to relationships between interface classes and/or interface operations. The product of this process is the Pattern-Level with Interfaces diagram, which is the refinement of the Pattern-Level diagram. Figure 14-17 illustrates the Pattern-Level with Interface diagram for the Client AE.

*Figure 14-17. A Pattern-Level with Interfaces diagram for the client side of a medical informatics application.*
The Receiver interface class of the CommandInterpreter instance receives the request from the Model interface class of the MVC instance. The command is then interpreted by the CommandInterpreter instance, and accordingly the Invoker interface class activate or send a message to the SOP instance using the AbstractFactory interface class. Results from processing the command by the server side components are forwarded from the SOP instance to the Model interface class of the MVC instance.

The AbstractFactory instance generates the necessary DIMSE messages, which could be communicated to the MessageCoordinator instance through the Mediator interface class. The MessageCoordinator instance determines the need for encoding and could send the DIMSE message through the Colleague interface class to Director interface class of the ProtocolMachine instance to generate the DICOM message. The Colleague interface class also communicates with the Entity_Interface class of the DICOM-UL instance to send messages to the server side components. The DICOM-UL instance sends messages that it receives from the server to the Mediator interface class of the MessageCoordinator instance.

**Constructing Detailed Pattern-Level Diagram**

To construct the Detailed Pattern-Level diagram, we express the internal parts of each instantiated pattern from the Pattern-Level with Interfaces diagram. Since we have used pervasive design patterns in developing the design for Client AE system, their structure can be found in the literature. The structure of the Builder, Mediator, Command, and Abstract Factory patterns can be found in [Gamma et al., 1995]. The structure of the Statechart pattern is found in [Yacoub & Ammar 1998b]. The MVC pattern details can be found in [Krasner & Pope 1998; Buschmann et al., 1996].

Figure 14-18 illustrates the Detailed Pattern-Level diagram for the client AE. The details of each pattern are shown. The design details of the ClientUserInterface pattern instance are hidden for simplification.

**Figure 14-18. A Detailed Pattern-Level for the client side of a medical informatics application.**
POAD Design Refinement for the Client Application Entity

The client AE is a complex subsystem. In the following we selected a portion of that subsystem to discuss in the design refinement phase. We limit our discussion in terms of the patterns to be instantiated and the services to be supported by the client.

Instantiating Pattern Internals

In this step we add application-specific nature to the Detailed Pattern-Level diagram by renaming internal pattern classes according to the application design environment, choosing names for pattern participants that are meaningful in the application context, and defining application specific names for operations in patterns.

Instantiating the internals of the SOP is illustrated in Figure 14-19. The SOP pattern instance is composed of

- **Service**. An abstract interface for all the services that could be requested by the client. Concrete service classes implement the Service interface class. For example, the Find, Move, and Get classes are concrete services.

- **IOD**. The IOD acts as the interface for all the information objects used in DICOM. Concrete information objects implement the IOD interface. For example, the Study, Patient, and CR-Image classes are concrete IODs. We note that DICOM categorizes the IODs as normalized and composite, which could be two refinement subclasses of the IOD interface.

- **SOPFactory**. The SOPFactory class is the interface for creating an SOP. An SOP contains a DICOM service of type Service interface and a DICOM IOD of type IOD interface. Concrete factories implement the SOPFactory interface for creating a service and an IOD. For example, the FindPatientSOP is a factory for creating the Find concrete service class and the Patient concrete IOD class. Similarly, the MoveImageSOP is a concrete factory for creating the Move concrete service class and the Image concrete IOD class.

Figure 14-19. Instantiating the SOP pattern.
Instantiating the internals of the ProtocolMachine is illustrated in Figure 14-20. The ProtocolMachine pattern instance is composed of:

- **MessageDirector**: The interface for the pattern instance, which constructs a DICOM message using one of the concrete message builders. It references the abstract interface for a message builder. MessageDirector is the Director class in design structure of the Builder pattern.

- **MessageBuilder**: The abstract interface for creating parts of a DICOM message. MessageBuilder is the Builder class in the design structure of the Builder pattern.

- **Concrete message builders**: Each concrete builder builds a specific type of message. For example, the C_Find-Builder builds a C_Find-Message, C_Get-Builder builds a C_Get-Message, and so on. The C_Find-Message, C_Get-Message, and so on are the Product classes in the Builder pattern.

**Figure 14-20. Instantiating the ProtocolMachine pattern.**
Instantiating the internals of the MessageCoordinator is illustrated in Figure 14-21. The MessageCoordinator pattern instance is composed of

- **Colleague.** The Colleague class is an interface for all colleagues. AColleague class knows it mediator.
- **MessageMediator.** The MessageMediator class defines an interface for communicating with colleague classes. DICOMMessageMediator is a concrete implementation of the MessageMediator interface.

**Figure 14-21. Instantiating the MessageCoordinator pattern.**

Developing an Initial Class Diagram

Starting from the Detailed Pattern-Level diagram in Figure 14-18, we use pattern interfaces and the instantiated details of pattern internals (as obtained in the previous subsection) to construct a UML class diagram. For simplification we consider only the portion of the design that contains the three pattern instances: SOP, MessageCoordinator, and ProtocolMachine. For those three instances, the interfaces are all of type class interface. Hence, we trace the internal classes that implement those pattern interfaces and create a class association between the two internal classes. Figure 14-22 illustrates the initial class diagram for the design.

**Figure 14-22. Portion of the initial class diagram design for the Client AE.**
From Figure 14-18, the AbstractFactory class interface for the pattern instance SOP has a dependency relationship with the Mediator interface class of the MessageCoordinator pattern instance. AbstractFactory interface is renamed to SOPFactory (Figure 14-19) and Mediator is renamed to MessageMediator (Figure 14-21) during the instantiation. Then, in the class diagram in Figure 14-22, the SOPFactory class has an association relationship with the MessageMediator class.

Similarly, from Figure 14-18, the Colleague interface class of the MessageCoordinator pattern instance has a dependency relationship with the Director interface class of the ProtocolMachine pattern instance and the AbstractFactory interface class of the SOP pattern instance. As a result, the Colleague interface class in the class diagram will have a class association with the SOPFactory class and the MessageDirector class, as shown in Figure 14-22.

Figure 14-22 illustrates the portion of the initial class diagram design containing classes generated from the three pattern instances MessageCoordinator, ProtocolMachine, and SOP.

It can be easily recognized that the patterns are still notable in the class diagram, as shown by the dotted boxes around the classes. Recall that we do not discard or swap away earlier diagrams. As part of POAD, all the models in Figures 14-16 through 14-22 are saved as analysis and design models. It is the role of a tool support to save these models and to provide the necessary traceability mechanisms between them.

**Design Optimization**

The class diagram obtained from gluing patterns together at the high-level design may not be dense or profound because we just strung the patterns together. It may have replicated abstract classes because we use multiple instances of the same pattern type. However, for this case study only one instance of each pattern type is used; hence, we do not have replicated abstract classes to eliminate.

We could also seek optimization by merging together classes from different patterns. From Figure 14-22, we notice that the interface class Colleague for the MessageCoordinator pattern instance of type Mediator has an association relationship with the two classes MessageBuilder and SOPFactory. These two classes are the actual colleagues communicating through the MessageMediator. Therefore, we could consider them the concrete colleagues in the MessageCoordinator pattern instance. As are result, the two classes MessageBuilder and SOPFactory inherit from the Colleague class, and the DICOMMessageMediator references the two classes instead of ConcreteColleague classes. Figure 14-23 illustrates a refined version of the partial class diagram of Figure 14-22.

**Figure 14-23. Refined class diagram for the Client AE subsystem.**
Summary

This chapter illustrated the applicability of the POAD process to develop a pattern-oriented design for large-scale applications such as medical informatics systems. Due to the large size of the application, we illustrate the analysis and design process for portions of the system. The case study illustrates that the POAD approach is applicable to a variety of applications in different domains and can scale from simple academic-style applications to complex, industrial-level systems. We also concluded how the POAD approach is easy to apply in designing systems in which architecture is based on components with complex behavior, where statechart and FSM patterns can be used to design those components.
Part V: Automation and Summary

In Part 5 we present some steps toward the automation of the POAD methodology. Chapter 15 discusses the UML metamodeling support for POAD models. Defining the relationship between POAD models and the UML metamodel is essential to enable tool support by existing UML tools. In Chapter 16 we discuss the existing tool support for applying, modeling, and composing design patterns and their support for POAD. Finally, Chapter 17 wraps the discussion on POAD and presents some plans to take POAD further into mainstream development.
Chapter 15. Relation to UML Metamodel and Specification

In this chapter we discuss the relationship between concepts used in POAD modeling and the UML metamodels. If you are using POAD to develop the design of an application, this chapter does not provide relevant information; you can refer to Chapter 7 to learn about the POAD development process. If you are developing a UML tool or a new UML-based development methodology, you might be interested in reading this chapter.
Syntax Is Not Enough

Many software designers and analysts find UML syntax sufficient to serve their modeling needs. When the main objective is to analyze and design applications using UML modeling constructs and model views, the syntax of the UML constructs is usually sufficient to serve that purpose. However, methodologists and tool developers are more interested in the formal aspects of the modeling languages, because they develop new methodologies and new modeling constructs that they want to formally tie to the existing modeling languages. They also want to understand the semantics of the modeling language rather than just using the exterior syntax. Tool developers are also interested in the semantics. When a tool is developed to support a modeling language, the tool should support not only the syntax and notation defined by the language but also the formal meaning of the modeling constructs. Moreover, the UML models developed by different tools should be interoperable. The only way to make these models interoperable is to make them refer to the same things with the same name—that is, follow the same modeling standard, a common underlying model, or a common ontology. This chapter is concerned with these common models for UML and POAD. If you are not developing a UML tool or a new UML-based development methodology, we encourage you to skip this chapter.

Modeling languages such as UML provide visual specifications in terms of model diagrams and views. Visual representation is not sufficiently expressive to define modeling constraints. Designers are motivated to supplement existing visual notations with constraints specified textually. Natural languages are not adequate for textual specification. Several OO design methodologies, such as Catalysis [D'Souza 1998], replace natural language by mathematical notation. Research in formal methods usually makes assumptions about the designer’s strong mathematical background. This is not usually the case, and hence most formal languages have problems with approachability. As a result, the UML consortium supplements UML visual notation with a formal language that is easy to apply, the Object Constraint Language (OCL).

The constructs used in UML as a modeling language are based on a common model that is called the UML metamodel [UML 2002]. The metamodel of a modeling language defines the meanings of the modeling language constructs and their relationships. The metamodel together with some constraints written in OCL are the basis of the UML semantics.

In the previous chapters we illustrated how to use the UML notation and extension mechanisms to develop the models used in the POAD methodology. In this chapter we take a closer look at the semantics of the modeling constructs used in POAD, such as the concept of a constructional design pattern, pattern interfaces, and pattern interface relationships. We discuss the relationships between the POAD modeling constructs and the underlying model behind the UML. The following discussion is not meant as a proposal to modify the constructs and rules of the UML semantics. Instead, the constructs and rules are used here for illustration and to explain the relationship between the modeling constructs used in POAD and those provided by UML.

The UML semantics [UML 2002] is defined along three dimensions: abstract syntax, well-formedness rules, and semantics. The abstract syntax dimension is used to describe the UML metamodel class diagram. The Well-Formedness Rules dimension specifies constraints on each of the elements described in the metamodel class diagrams. The semantics dimension is the informal natural language description that defines the meaning of the model elements and their relationships. The OCL language is used to define the well-formedness rules of the UML semantics.

In the following section, we briefly introduce the formal language underlying the UML semantics. Then, we introduce some metamodel elements and discuss their relationship to the UML semantics along the three dimensions abstract syntax, well-formedness rules, and semantics.
The Object Constraint Language

What Is OCL?

The OCL is a primary contribution by IBM [OCL 1999b] to the UML specifications. J. Warmer and A. Kleppe developed OCL as a language for business modeling within IBM. It is used within UML both to help formalize the semantics of the language itself and to provide a facility for UML users to express precise constraints on the structure of their models [Warmer & Kleppe 1999]. UML defines common constructs for OO modeling, such as class models, object models, statecharts, and use cases. OCL is the part of UML that can be used to specify all kinds of constraints, preconditions and postconditions, and guards over the various model elements.

OCL can be considered an expression language, a modeling language, and a formal language [OCL 1999b]. As an expression language, an OCL expression is guaranteed to be without side effects; that is, it does not change anything in the model element values. Therefore, the system does not change when an OCL expression is evaluated. An OCL expression can be used to specify changes in the system (e.g., using a postcondition) but does not change the state itself. All values for all objects, including all links, remain unchanged.

As a modeling language, OCL is used to specify the model elements. It is not a programming language; thus, it cannot be used to specify the program control flow or logic. For instance, we cannot invoke processes or activate nonquery operations within OCL. As a modeling language, all implementation issues are out of scope and cannot be expressed in OCL.

As a formal language, every construct has a formally defined meaning. The specification of OCL is part of the UML specification [OCL 1999a; OCL 1999b]. OCL is not intended to be a general-purpose formal language, like VDM, Z, and others. Instead, it is very specific for formally specifying and constraining OO models and metamodels.

Using OCL

Using OCL in defining constraints on UML models and metamodels has several benefits:

- In OO modeling, a graphical model, such as a class or an object model, is not usually sufficient for precisely specifying the model without ambiguities. To unambiguously specify the design models, there is a need to describe additional constraints and guarding conditions about the model elements. Specifying constraints in natural language has always resulted in ambiguities. To write unambiguous constraints, a formal language has to be used. For OO models, OCL is developed for this purpose.

- Traditional formal languages often require strong mathematical background to understand and use. This is difficult for average system designers and modelers. OCL has been developed to bridge this gap by providing a formal language, which is easy to read and write.

- OCL can be used for a number of different purposes:
  - Specifying invariants on classes and types in the class model.
  - Describing preconditions and postconditions on operations and methods.
  - Specifying the metamodel elements of the UML metamodel class diagrams.
In the following sections we use OCL constructs to specify the well-formedness rules of the model extensions that we use in POAD and their relationship to UML metamodel. For details about OCL constructs and grammar, refer to *The Object Constraint Language: Precise Modeling with UML* [Warmer & Kleppe 1999] and to the OCL specifications (1999a; 1999b).
Adding POAD Constructs to the UML Metamodel

The UML semantics document is primarily intended as a comprehensive, precise specification of the UML semantic and modeling constructs at the metamodel layer. The UML metamodel is defined as one of the four-layered metamodeling architecture:

- **User data.** User data defines specific information in the application domain. For example, when we are developing a trading application for a particular company, we identify specific user data, such as the orders users have placed, the items they ordered, and the manufacturers from which they ordered. User data represents specific instances of the application model.

- **Model.** The model defines a language to describe an information domain. For instance, when we develop a model for a trading application, we identify the classes used in a trading system, such as the Order or Trader classes. The model is an instance of a metamodel layer.

- **Metamodel.** Metamodel defines the language for specifying a model. For example, to develop a metamodel, we need to define things that we will use to develop a model. For UML, we use class, attribute, operation, component, package, and so on in modeling applications, and hence the UML metamodel captures these elements and describes the relationship between them.

- **Meta-metamodel.** This is the infrastructure for a metamodeling architecture that defines the language for specifying metamodels. For example, to describe the relationships between the metamodel elements such as class, attribute, operation, component, and package, we need a representation language. UML uses metaclasses and meta-attributes, which have the same syntax as UML class diagram models.

POAD constructs are model constructs that the designers use to develop pattern-oriented application models, and hence they are metamodel constructs. Just as classes, operations, and packages are metamodel elements, patterns, pattern interfaces, and so on are considered metamodel elements as well. In the following sections we tie POAD constructs to UML metamodel constructs using the same meta-metamodel elements that UML uses: the MetaClasses and MetaAttributes. It is not the objective of this chapter to describe UML metamodels; instead, we tie the POAD modeling constructs to the current UML metamodel constructs. We encourage you to become familiar with the UML semantics part of the UML specification [UML 2002] before proceeding with the rest of this chapter. Most of the following discussion is more focused on POAD metamodel constructs rather than on an explanation of the existing UML metamodel elements.

The UML metamodel layer is decomposed into three specification packages: Foundation, Behavioral Elements, and Model Management. Figure 15-1 illustrates the high-level decomposition of the UML metamodel packages and their constituting packages. Each package contains the UML metamodel classes and their relationships in a class diagram.

![Figure 15-1. The UML metamodel packages.](image-url)
The Foundation package is the infrastructure for UML. It contains the Core, Extension Mechanism, and Data Types packages. The Behavioral Element package describes the superstructure for behavioral modeling in UML. It contains Common Behavior, Collaboration, Use Cases, State Machines, and Activity Graphs packages. The Model Management package specifies how model elements are organized into models, packages, and subsystems. Each package is described using graphical notation (abstract syntax), formal language (well-formedness rules), and natural language (semantics).

- **Abstract syntax.** A class diagram model together with supporting natural language is used to describe the abstract syntax.
- **Well-formedness rules.** These rules specify constraints using the UML-adopted formal language, the OCL. In addition, natural language is used to describe the natural meaning of the constraint. OCL rules are used to specify the static semantics of the language—that is, how an instance of a construct should be connected to other instances in a meaningful context.
- **Semantics.** Natural language is used to define the dynamic semantics of the metamodel—that is, the meaning of a well-formed construct. It describes only concrete metaclasses.
Abstract Syntax

In UML, the abstract syntax of the metamodel is represented using a set of class diagrams defining the metamodel constructs (metaclasses) and their relationships. A metamodel class diagram specifies some well-formedness rules, such as the multiplicity of a relationship, navigation direction, ordering, and some constraints. The metamodel class diagram is followed by a description of the classes, their attributes, and the role names of the relationships with other metaclasses.

Figures 15-2 to 15-7 illustrate parts of the UML metamodel class diagrams that we augment with additional POAD constructs. These parts are taken from various UML metamodel packages and their class diagrams. Other UML metamodel class diagrams are not shown. The model elements that we add are shown in light gray color. POAD constructs are related to the following class diagrams:

1. The Model Management package. The ModelManagement class diagram is augmented with the Pattern class.
2. The Foundation package. Inside the Core package, the Dependencies class diagram is augmented with the PatternDependency and InterfaceBinder classes.
3. The Foundation package. Inside the Core package, the Relationships class diagram is augmented with the PatternAssociation relationship.
4. The Foundation package. Inside the Core package, the AuxiliaryElements class diagram is augmented with the PatternInfo class as an instance of the Comment class. The presentation of a pattern is a subclass of the PresentationElement.
5. The Foundation package. Inside the Core package, the Classifiers class diagram is augmented with the PatternInterface classifier and its association relationship to the Class and Interface classifiers.
6. The Foundation package. Inside the DataType package, we have added new data type classes, such as PatternKind.

Figure 15-2. The extended class diagram for the Model Management package.
A pattern, as used in POAD, is defined as a constructional design pattern whose solution is a set of collaborating classes. Constructional design patterns are further explained in Chapter 4. A constructional design pattern is modeled here, similarly to packages, as a grouping mechanism for model elements.

A pattern as a metamodel class in the UML metamodel is illustrated in Figure 15-2 as part of class diagram for the UML Model Management package. A pattern is subclassed from a namespace. A namespace is a part of the model that contains a set of model elements whose names designate unique elements within that namespace. A namespace is an abstract metaclass. It is a ModelElement that owns other ModelElements. Each Model Element is owned by, at most, one namespace.

A pattern is a namespace for its owned elements. In POAD the owned elements of a pattern are constrained to classes and their relationships, pattern interfaces, and pattern interface binders. The pattern, as a grouping element, exposes its internals and interfaces according to the design-level abstraction. At a high level of abstraction, the internals of the pattern are invisible—only the interfaces are visible for integration. At a lower abstraction level, the internal classes of the pattern become visible to other model elements for the purpose of integration and merging with other classes in the application design.

A pattern, in principle, should be simple and should not attempt to solve more than one well-defined problem. Hence, in most cases a pattern is not a generalizable element and cannot be part of other patterns, since generalization in many cases implies addition or refinement of a common principle. While packages can be nested, patterns cannot. This is one difference between the semantics of a package and a pattern.

In POAD, parts of a pattern should not be imported or exported in the sense used in packages. A package usually owns its element and can allow other packages (or constructs) to access its elements. A pattern doesn't have such restriction, since an application class will eventually be the integration and merge of various parts of other patterns. This is another difference in the semantics of packages and patterns.

Though deep down the semantics of packages and patterns may differ, the end user may not be concerned about these differences and would use package syntax to represent a pattern as a grouping mechanism, as illustrated in chapters 5 and 6.

Attributes

Each pattern instance has a type. We refer to a Type as the well-known and documented name of an abstract pattern—for example,
Observer, Factory, and Strategy. This attribute is of type PatternKind, which is defined as an extension to the DataType metaclass diagram as a stereotyped enumeration.

PatternDependency

PatternDependency is illustrated in Figure 15-3 as a metamodel class that is added to the Dependencies class diagram of the UML Semantics Core package. A Pattern Dependency inherits from the general Dependency class. A PatternDependency is a specialized type of Dependency in which the client and supplier are of the same type, Pattern. A PatternDependency indicates a semantic relationship between two patterns—a situation in which a change to the source pattern may require a change to the target pattern or a pattern delegates the processing of a certain function to the other. This type of relationship is used in POAD at the high-level design phase where the system is modeled as a composition of design patterns and PatternDependency relationships between those patterns. At a lower design phase, each PatternDependency relationship is further refined into PatternAssociation relationships between pattern interfaces. A PatternDependency indicates a semantic relationship of the type usage. The usage dependency class can be used to capture the semantics of a pattern relationship.

**Figure 15-3. The extended Dependencies class diagram from the Core package.**

Associations

Since a PatternDependency inherits from Dependency, it has a client and a supplier.

*Client:* The pattern that requests service from the supplier. It represents the Model Element at the starting end of a directed arrow representing the dependency.

*Supplier:* The pattern that provides service to the client pattern. It represents the ModelElement at the terminating end of a directed arrow representing the dependency.
**PatternAssociation**

A PatternAssociation defines a semantic relationship between the interfaces of two patterns. A PatternAssociation is illustrated in Figure 15-4 as a metamodel class that is added to the Relationship class diagram from the Core package. A PatternAssociation is a refinement of a PatternDependency in which the usage relationship between two patterns is further analyzed and association relationships are identified between the two pattern interfaces. The instances of an association are a set of tuples relating instances of the PatternInterfaces. A PatternAssociation inherits from the Association metaclass. It has exactly two AssociationEnds. Each end is a pattern interface, which could be an operation interface or a class interface.

**Figure 15-4. The extended Relationship class diagram from the Core package.**

![Diagram](image)

**Associations**

*Connection.* A PatternAssociation consists of two AssociationEnds, each of which represents a connection of the association to a PatternInterface.

**PatternInfo**

A PatternInfo is a subclass of the Comment class in the AuxiliaryElements class diagram from the Core package, as illustrated in Figure.
15-5. It is a specific type of documentation that reflects the documentation details of a design pattern. The documentation of a pattern takes several formats, such as the Alexandrian, GoF, or POSA templates. Some of the sections that could be documented about the pattern include the forces for using a specific pattern, the consequences of applying the pattern, the intent and motivations, and a programming template for its instantiation. These sections could be reflected as attributes of the PatternInfo class defined by the appropriate corresponding data types.

Figure 15-5. The extended AuxiliaryElements class diagram from the Core package.

PatternInterface

A PatternInterface is a set of classes and methods used to define how the pattern interfaces to other patterns. As illustrated in Figure 15-6, PatternInterface is added as a metamodel class in the Classifiers class diagram from the Core package. PatternInterface is a specific type of interface for constructional design patterns. To qualify as a design component, a pattern has to have interfaces. The interface helps in gluing patterns together at the high level of abstraction and hiding internal details and design decisions made within the pattern. Pattern interfaces are defined in terms of interface methods and interface classes (Chapter 4). Therefore, a PatternInterface has one or more classes and UML interfaces (operations). A pattern, as a namespace, owns some elements; one of the elements it owns is its interface metaclass, which defines the classes and operations used as pattern interfaces.

Figure 15-6. The extended Classifiers class diagram from the Core package.
Associations

*OperationInterface*: A set of operations used to model the pattern interfaces. These operations are parts of the internal structure of the pattern defined and implemented by methods in one or more of the pattern’s internal classes.

*ClassInterface*: A class that defines the interface of a pattern. This class is one of the internal classes of the pattern.

PatternKind

In the metamodel, PatternKind is a data type that defines an enumeration whose values are the types of design patterns. The PatternKind construct is added in the DataTypes class diagram from the DataTypes package, as illustrated in Figure 15-7. Usually, this is an enumeration of the design pattern names. Its values denote the type of an instantiated pattern. Examples of the enumeration values are Composite, Strategy, State, Observer, Blackboard, Reactor, and Dispatcher.

PatternPresentation

The UML semantic has a very flexible metamodel class diagram design that accommodates specific representations of model elements and their notation. For each Model Element, we can specify a PresentationElement, as illustrated in Figure 15-5. This separation of modeling and notation provides a very flexible approach for tools to separate between the implementation of the semantics and the presentation of the notation. For POAD, we can attach a presentation for a pattern as a PatternPresentation class, which inherits from the PresentationElement. The UML package and interface representations are used in this book as discussed in Chapter 5.

InterfaceBinder

An InterfaceBinder is a specific type of Dependency that is used to connect the PatternInterface to the corresponding internal elements of a pattern. InterfaceBinder is added to the Dependencies class diagram from the Core package metamodel, as illustrated in Figure 15-3.
inherits from the general Dependency relationship. The semantics of this dependency relationship is trace, where the pattern interfaces in terms of interface classes or interface operations are traced to the internal classes of the pattern. The Abstraction Dependency relationship with trace semantics can be used to achieve the same objective of InterfaceBinder. Depending on the visual syntax used to represent pattern combinations at different levels (see Chapter 5), this binding manifests itself into different forms, which might be a graphical link between the interfaces and the pattern content or a coloring schema to illustrate which parts of the pattern content are the interfaces. For the visual models that POAD uses in Chapter 5, we use an explicit link.
Well-Formedness Rules

Well-formedness rules specify the static semantics of each construct in UML in terms of a set of invariants of the metaclass instance. These rules specify constraints over the attributes and associations in the metamodel. Each invariant is described in natural language (informal) together with an OCL expression (formal). In the following, we describe the rules for the additional constructs identified in the previous section.

Pattern

1. A pattern (as used in POAD) can only contain Classes, Associations, Generalization, Dependency, Constraints, PatternInterface, or PatternBinder.

   \[
   \text{self.contents->forAll ( c | c.oclIsKindOf(Association) or c.oclIsKindOf(Generalization) or c.oclIsKindOf(Dependency) or c.oclIsKindOf(Constraint) or c.oclIsKindOf(Class) or c.oclIsKindOf(PatternInterface) or c.oclIsKindOf(PatternBinder) )}
   \]

2. No two ModelElements have the same name inside a pattern.

   \[
   \text{self.contents->forAll( c1,c2 | c1.name = c2.name implies c1=c2 )}
   \]

3. Pattern nesting is not allowed because a pattern cannot be a part of another pattern. We note that this constraint is covered in constraint number 1, but we repeat it here for clarity.

   \[
   \text{self.contents->forAll ( c | c.oclType <> Pattern )}
   \]

4. A pattern cannot be a generalization or a refinement of another pattern.

   No OCL Expression

5. A pattern is not an instantiable programming construct in an implementation language.

   No OCL expression

6. A pattern should have at least one PatternInterface to qualify as a design component.

   \[
   \text{self.contents->exists( c | c.oclIsTypeOf(PatternInterface))}
   \]

7. A dependency relationship between two patterns is only of type PatternDependency.

   \[
   \text{self.clientDependency->forAll( c | c.oclIsTypeOf(PatternDependency))}
   \]
   \[\text{and}
   \]
   \[
   \text{self.supplierDependency->forAll( c | c.oclIsTypeOf(PatternDependency))}
   \]

Additionally, the operation contents on a pattern results in a set containing the ModelElements owned by the pattern.

\[
\text{contents : Set(ModelElement)}
\]
\[
\text{contents = self.ownedElement}
\]
**PatternDependency**

1. The stereotype of dependency between patterns is usage.
   
   ```
   No OCL Expression
   ```

2. The client of a pattern dependency is a pattern.
   
   ```
   self.client->oclIsTypeOf(Pattern)
   ```

3. The supplier of a pattern dependency is a pattern.
   
   ```
   self.supplier->oclIsTypeOf(Pattern)
   ```

4. A pattern dependency is only between two elements.
   
   ```
   self.client->size = 1 and self.supplier->size = 1
   ```

**PatternAssociation**

1. The AssociationEnds of a PatternAssociation must have a unique name.
   
   ```
   self.allConnections->forAll( c1,c2 |
   c1.name =c2.name implies c1 =c2 )
   ```

2. PatternAssociation is only between two elements.
   
   ```
   self.allConnections->size = 2
   ```

3. The type of AssociationEnds of a PatternAssociation is either a UML class or a UML interface.
   
   ```
   self.allConnections->forAll( c |
   c.type.oclIsTypeOf(Class) or
   c.type.oclIsTypeOf(Interface) )
   ```

4. A PatternAssociation is not generalizable.
   
   ```
   self.isLeaf = true and self.isRoot = true
   ```

5. At most, one AssociationEnd is an aggregation.
   
   ```
   self.allConnections->select(aggregation > #none)->size <= 1
   ```

6. If one AssociationEnd in a PatternAssociation is an aggregation, it is of type Class.
   
   ```
   self.allConnections->select(aggregation <> #none)->forAll ( c |
   c.type.oclIsKindOf(Class) )
   ```

7. The connected interface classes or methods of the AssociationEnds should be included in the namespace of the PatternAssociation.
   
   ```
   self.allConnections->forAll ( c |
   self.namespace.allContents->includes(c.type)
   ```

8. A PatternAssociation has no association class.
Additionally, the operation `allConnections` results in the set of all `AssociationEnds` of the `Association`.

```plaintext
allConnections : Set(AssociationEnd);
allConnections = self.connection->union(self.parent->select
(s | s.oclIsKindOf(Association))->collect (a : Association |
  a.allConnections))->asSet
```

**Note**

We have not considered the Navigation of the `PatternAssociation` for `PatternInterface`. Navigation can include Provided/Required operation interfaces or Client/Supplier class interfaces.

**PatternInterface**

1. A `PatternInterface` should not contain any other `ModelElement`.
   ```plaintext
   self.allContents->isEmpty
   ```

2. For each method in a `PatternInterface`, the namespace of the pattern must have a matching method.
   ```plaintext
   self.operationInterface->forAll( i | i.allFeatures->Select(f | f.oclIsKindOf(Operation))->forAll(op |
   self.namespace->includes( cc |
   cc.oclIsTypeOf(Class) and cc.namespace->includes( cc_op |
   cc_op.oclIsTypeOf(Method) and cc_op = OP)))))
   ```

3. For each class in a `PatternInterface`, the internal contents of the pattern (pattern classes) must have a matching class.
   ```plaintext
   self.classInterface->forAll( i | self.namespace->includes( cc |
   cc.oclIsTypeOf(Class) and cc = i ) )
   ```

4. Every element in the interface should be connected to some internal parts of the pattern. This is satisfied if constraints 2 and 3 are satisfied.

**InterfaceBinder**

1. The type of the client and supplier of an `InterfaceBinder` is the same because it is just a connection mechanism to relate pattern internals to the pattern interfaces.
   ```plaintext
   self.client.oclIsTypeOf(Class) implies
   self.supplier.oclType = Class
   self.client.oclIsTypeOf(Interface) implies
   self.supplier.oclType = Interface
   self.supplier.oclIsTypeOf(Class) implies
   ```
self.client.oclType = Class
self.supplier.oclIsTypeOf(Interface) implies
    self.client.oclType = Interface

2. The type of client and supplier is either Class or Method.

    self.client.oclType = Class or self.client.oclType = Interface
    and
    self.supplier.oclType = Class or self.supplier.oclType = Interface

3. An InterfaceBinder should belong to the namespace of an enclosing pattern.

    self.namespace.oclType = Pattern

4. The connected client and supplier should be included in the namespace of the InterfaceBinder, which is restricted in the previous constraint to patterns only.

    self.namespace.contents->includes(client.type)
    and
    self.namespace.contents->includes(supplier.type)

[ Team LiB ]
Semantics

In this section we describe, using natural language, the meaning of the modeling constructs used in POAD.

Pattern

A pattern is not a classifier In POAD we use a constructional design pattern as a design component, which encapsulates classes and their relationships to solve a particular design problem. A classifier is a construct that has behavioral and structural features such as a class, a component, a data type, an interface, and others. A structural feature refers to a static feature of the model, such as an attribute. A behavioral feature refers to the dynamic feature of a model element, such as an operation or a method. All different kinds of behavioral aspects of a classifier, such as operations and methods, are subclasses of BehavioralFeature. A pattern does not possess its own independent structural or behavioral properties. For example, we do not assign attributes to a pattern as a modeling element, and a pattern does not possess its own operations or methods. The behavioral and structural aspects of the pattern are defined and captured by its internal design—that is, the class and collaboration design diagrams of the pattern internals—and no additional structure or behavior is introduced by the pattern as a modeling construct.

A pattern is not a package In a UML metaclass diagram, a package is a collection of ModelElements, whatever the type of elements included in the package. A package is generalizable and can have children and parents in an inheritance tree. A package can export parts of its internal ModelElements and can import some ModelElements from other packages. On the contrary, for POAD, a pattern is a collection of classes only. Patterns cannot be parts of other patterns, and patterns cannot inherit from other patterns. Patterns do not export or import parts of their internal structure. Participants from various patterns can be merged into one application class.

A pattern is not a subsystem In UML, subsystems are specialized types of packages, and hence they cannot be considered patterns. A pattern, as presented in POAD, is a collection of classes that solves a specific design problem. It is a namespace for a collection of classes, which offers interfaces in terms of methods and classes.

Every constructional design pattern has an interface by which it is glued to other patterns. A PatternInterface is defined in terms of operations and classes. These operations and classes are internal parts of the pattern that are exposed to the external modeling environment. A pattern interface is related to internal ModelElements of the pattern using InterfaceBinder.

Relationships

The relationships between patterns are dependencies, which we specify as PatternDependency. These types of dependencies are used to model the usage relationships between the client and the supplier. PatternDependency specifies the usage relationship between two patterns. A PatternDependency is further refined to Pattern Association. A PatternAssociation is the relationship between the interfaces of two related or collaborating patterns. A PatternAssociation can represent the relationship between two interface classes, two interface operations, or an interface class and an interface operation. Whereas PatternDependency is a usage relationship between two patterns, PatternAssociation is an association relationship between the two PatternInterfaces.

InterfaceBinders are another type of dependency relationship, which are used to connect PatternInterfaces to the internal parts (classes or methods) of the patterns. They are used as binding relationships with the semantics of trace, where we trace the interfaces to the internals of the pattern.
Summary

There are several ways by which the elements that serve the POAD methodology can be incorporated in UML. The simple and easy way is to use UML syntax and UML extension mechanisms illustrated in chapters 5 and 6. In this chapter we presented another way that relates the POAD modeling concepts to the UML metamodel. The model constructs discussed in this chapter are useful for tool developers and methodologists who are more interested in the formal aspects of the models and their semantics. For application design and development, you need not worry about the concepts discussed in this chapter.
Chapter 16. Tool Support for Designing with Patterns

The Need for Tool Support

Pattern Tools

Requirements and Features of a POAD Tool

Summary
The Need for Tool Support

Gluing constructional design patterns to develop large application designs is a difficult task that requires resolving many integration issues. The POAD process is iterative—a feature that requires the designer to preserve the models produced in each and every development step. Tool support for the development process facilitates the analysis and design steps.

The designer might consider using object-oriented modeling tools to support the development process. We might consider this a possibility because (a) the end product is an OO design, so an OO modeling environment will be suitable; (b) the models used in POAD are mainly UML design models; and (c) there are many commercial tools that support UML, OO modeling, and OO development processes.

However, many OO modeling tools do not explicitly support patterns as design components with interfaces. Moreover, there are several design activities required by the POAD process that must be supported by a POAD-enabled development environment.

A development methodology is often more popular if it is supported by a development environment. Tools help analysts and designers conduct the various development activities within the development process. Tool support for POAD is no exception. POAD tools facilitate the development of pattern-oriented designs and frameworks. Gluing patterns together in a design is not just a simple process of interconnecting the internal components of the patterns. There are several challenges that make tool support essential, including the following:

- In POAD, we have to merge participants from various components. This may cause some issues, such as keeping track of patterns and making them explicit in the final design.
- POAD produces models at various design levels. POAD design models can be viewed at different levels of abstraction. At a high level of abstraction, the design is viewed as patterns and their dependency relationships. At a lower level of abstraction, the internals of the patterns are revealed to produce traditional OO class diagrams. Tool support is essential to facilitate the traceability between the design elements used in these different design levels.
- In POAD the notion of pattern interfaces is made explicit in the design models to integrate and glue constructional design patterns. Design tools are required to support the notion of pattern interfaces and using a pattern as a design component.
- Current visual modeling languages and their tool support do not explicitly incorporate the concepts of pattern-level diagrams and pattern interfaces. The POAD tool should support high-level design models in terms of pattern diagrams and should integrate with existing tools that support lower-level design models in terms of class diagrams.
- POAD tools should support the design process steps that we discuss in Chapter 7 and the visual models that we discuss in Part 2. POAD tools should provide support for Pattern-Level, Pattern-Level with Interfaces, and Detailed Pattern-Level models.
- For large, complex applications, the design should be manageable at various levels of abstraction.

In this chapter we discuss the features required in a POAD-enabled tool that uses constructional design patterns as design components. We first discuss existing tools that support designing applications with patterns. We then discuss how a visual composition tool can support the POAD approach and the essential requirements in a POAD-enabled tool.
Pattern Tools

Over the years, much effort has been expended on tool support for documenting and instantiating patterns. We briefly discuss the approach used in some these tools and identify their limitations.

Framework Adaptive Composition Environment

For code generation and pattern instantiation purposes, the software development environment FACE (Framework Adaptive Composition Environment) [Meijler et al. 1997] guides the process of instantiating patterns to develop software applications. FACE is an environment mainly targeted for explicitly representing design patterns in the application design to solve what is called design-implementation gap.

The design-implementation gap refers to the inconsistency between the design pattern at a high abstraction level and the code implementing the pattern at the implementation level. In an incremental development lifecycle, the source code generated from a design pattern is usually changed at the implementation phase, and these changes break the implicit link between the higher abstraction level and the lower-level implementation. Supporting incremental development lifecycle without abandoning the higher-level design pattern abstraction is one of the main objectives in the development of FACE. FACE is an approach that bridges this design-implementation gap by supporting incremental development using frameworks at the abstraction level of design patterns.

FACE is based on defining a schema that is used to capture the classes and the roles they play in the pattern; operations and the roles they play in the pattern; relationships between classes and/or operations; and parameters. The user starts with a primary schema containing the abstract classes of a pattern and their associations. The primary schema is extended by defining concrete classes with associated roles and operations, creating corresponding relationships, and specifying necessary parameters.

The schema used in FACE is a graphical syntax that resembles the OMT class diagram for a pattern. In addition, the schema represents some pattern-specific relationships, such as operations being explicit model elements, and relationships between operation and classes that illustrate which operation instantiates which class. The development environment allows the designer to create application-specific details in terms of concrete classes and operations. The development environment claims to close the gap between design and implementation, since it captures implementation-level details such as operations and how operations create or invoke other operations at the design level. The FACE development environment captures enough information for running the application at the design level. It represents a "modeling = programming" approach that is attributed to the use of a schema (model) that makes all the information required by the application developer explicit at the design phase. A visual composition tool can help the developer to create and adapt these models correctly according to pattern specific syntax rules.

The main drawback of FACE is that it uses its own schema for modeling the pattern structure. Though the schema is closer to OMT notation, it has its own elements. Figure 16-1 shows a snapshot from the models (schema) used in FACE to model the AbstractFactory pattern.

Figure 16-1. A Primal-Schema and MetaSchema for the Abstract Factory pattern.

We believe that the UML modeling language should be used in any OO modeling tool, since it is the integration of many OO syntax and semantics. It is unlikely that software designers will abandon the de facto standard for modeling and use a proprietary modeling syntax and semantics. It is also obvious that the design-implementation gap that FACE addresses is now solved by many reverse-engineering features in OO modeling tools, such as Rational Rose [Rational 2002] and TogetherSoft [TogetherSoft 2002]. Such modeling tools allow synchronization between the implementation and the design models. Therefore, we do not see a direct support for the POAD process by the FACE development environment.

Fragmentation Technique

The fragmentation technique and its development environment [Florijin et al. 1997] is an approach for binding patterns together with one another and with other non-pattern design elements. This technique mainly uses the solution structure of the pattern for composition, where a pattern is mostly described in terms of an abstract design structure expressed in classes, methods, and relationships.

The fragmentation technique finds the description of a pattern solution in terms of structure diagram not sufficient to capture all aspects of the pattern. As a result, it refines the pattern design into smaller fragments that are used to capture such relationships. The fragment development environment is based on a single-level system that treats all design constructs as collections of fragments. A fragment represents a design element, which could be a class, a method, an association, and so on. Fragments can contain references to other fragments and can have roles associated with them. A design pattern class model is transformed into fragment representation. Other design elements that are not patterns are also transformed into fragment representation. A special graph notation is used to represent fragments and their relationships. For example, the fragment model for the MyObserver pattern instance of type Observer pattern is shown in the [Figure 16-2].

Figure 16-2. A fragmentation model.

A fragment browser is used to view, edit, and connect fragments together. Details about each fragment and the role it plays can also be displayed. The development environment also uses an OMT tool, which provides an OMT design-level view of the application. The designer can use the OMT tool to perform editing on the application design. The tool also supports for instantiating and binding patterns into the program. The OMT views list the patterns used in the application design and provide tools to identify roles that belong to different pattern instances.

The fragmentation tool provides three integrated views of an application: the code, which is expressed in terms of classes and methods; the design view, which is the abstraction of the code in addition to some design information and relationships; and occurrences of design patterns in the application. The fragmentation tool assists designers and the developers in three ways. It assists in generating code elements such as classes and class hierarchies through instantiation of a pattern that is retrieved from a collection of template patterns. The tool is also used to integrate pattern occurrences with the other parts of the application by providing means to bind application design elements to a role in a particular pattern. The tool achieves this binding using the fragmentation graph to indicate that an existing class plays some role as a participant in a pattern instance. The tool is also used to validate that a pattern is used and instantiated correctly in the application by checking that the design still meets the invariants governing the patterns.

The design of the tool relies on a refactoring package that allows the designer to use the tool in forward and reverse engineering. In forward engineering, pattern templates are used to compose the decimated (fragment) model of the pattern and integrate the fragments with existing application fragments. For reverse engineering, the tool can be used to document occurrences of patterns in existing applications and to modify the application to use the solution provided by the pattern.

The goal of the fragmentation approach is very similar to the goal of the POAD methodology: to introduce patterns as first-class citizens in an integrated CO development environment. The way the fragmentation technique works is to get the user (designer/analyst) involved at the fragmentation level to specify how things are tied together and how a pattern role is played by some other application class. Even at the OMT diagram level, the designer still hooks the roles together using a fragment view. While POAD shares the same principle of treating patterns as design constructs, we find it difficult to involve the designer at the fragmentation level. The designer should be able to do all the merging and role assignment activities using the UML models of the patterns and the application.

The fragment model is particularly useful as a backend engine to integrate the roles played by various design elements. The
fragmentation model appears as another non-UML representation that represents all design elements at one level. An advantage of using UML models is that we can have the design and code within the same integration environment. For example, there are many UML tools that do reverse engineering and synchronize the code with the design.

The fragmentation development environment does not support the visualization of the application design as pure a composition of pattern instance. This feature is required in POAD to represent the Pattern-Level and the Pattern-Level with Interfaces diagrams. Another limitation in the tool is that it is mostly language-dependent, since it was originally introduced and applied for the development of SmallTalk applications.

The advantage of this approach is that it works for integrating design pattern elements with other parts of the system that are not design patterns. However, this binding is mostly done using the fragmentation graph, which is a one-level system. We believe it will be difficult to scale up for large designs because of complexity introduced when the number of fragments is large. The tool support for pattern-oriented design should solve this problem at a higher level of abstraction.

**PSiGene**

The Pattern-Based Simulator Generator (PSiGene) development environment [Schuetze et al. 1997] was developed within the MOOSE (Model-Based Object-Oriented Software-Generation Environment), which is a framework for model-driven code generation for software applications. The project was mainly focused on providing support for control system designs in testing their applications through simulation before deployment in building and construction.

PSiGene is considered a system for partial formalism of design patterns as well as for code generation from patterns. The PSiGene CASE tool is a development environment for binding patterns from a predefined catalog with class models to construct the application design. It consists of a central database for different types of models, a set of model editors, and code generators for different types of software components. The process used in PSiGene is a domain-specific software development method based on design patterns and code-generation techniques. Though the approach and the tool were used for the domain specification of building simulators, the creators of the development methodology claim that the environment can be used with other applications.

In this domain-specific environment, patterns are used to describe how the simulation objects interact with each other. Therefore, design patterns are used as the glue between the simulation objects. A code template for every simulation pattern is saved in a code template catalogue and is used to generate the final application code.

In PSiGene a class model is used to describe the structure of the application. The glue between classes in the structure model is provided by ordinary class relationships, such as inheritance or aggregation, or by a design pattern, which captures the relationships between various class participants. The simulation patterns are used to capture the interaction between objects and encapsulate fragments of the simulator's functionality. They also contain code templates for the generation of simulation operations.

To use PSiGene, the designer creates the class model for the application using the class model editor provided in the modeling environment. The designer can also use a class library to select from a set of classes that he can use in modeling the application. After constructing the class model, the designer selects from a set of predefined simulation patterns. Those patterns are used to define the simulation problem. The designer also creates what is called a binding, which is used to map pattern elements to existing elements in the class model. The application of a pattern may trigger the user to define some other classes or methods, since a pattern can contain more elements (classes or methods) than those originally defined in the application class model. Therefore, the user must do several iterations between defining elements in the class model and applying specific simulation patterns. The tool then integrates and links everything together to produce the application model. The application code is generated by applying the code templates attached to each pattern and adding the simulation pattern methods to the appropriate classes. This code generation is done automatically, using the tool.

The main context for this approach is the application generators for the particular domain of simulating control designs for buildings. The patterns used within this environment are very domain-specific. The approach looks more like a refactoring approach in which the designer creates an initial design of the application, using a set of classes retrieved from a domain-specific class library or kernel, and then applies patterns to refactor the design. In this method patterns are mainly used to adapt the behavior of interaction between the classes according to a pattern solution. It is clear that this environment does not support patterns as building constructs or as first-class modeling elements, because it is mainly a refactoring and code-generation environment. The method is specific for building simulators. It generates the classes and methods for specific patterns from a catalog but doesn't interconnect patterns at the design level.
**Code Generation**

The research team at IBM [Budinsky et al. 1996] developed a tool for code generation from design patterns. The tool is built on the premise that a design pattern—as documented in the literature—is a design solution and not a code or implementation fragment. Therefore, each time the designer uses a specific pattern, he generates code by himself, and hence the code-generation effort is repeatedly duplicated. Each pattern has multiple implementations, and hence a code-generation tool is required to capture possible pattern implementations.

This tool provides the designer with a means to instantiate a design pattern into the application design. The tool addresses two main issues in instantiating a pattern in an application: (a) a means to define application-specific parameters and names that should be used when instantiating the pattern, and (b) a means by which the designer can select between tradeoffs in the pattern implementations and map those tradeoffs into various implementations.

The tool has a repository of design patterns at its backend; those patterns are mainly from the GoF collection. The web-based representation of the pattern catalogue helps the designer navigate through the pattern catalog and easily jump from one pattern section to another.

The tool allows the designer to choose between implementation tradeoffs. Each pattern type is associated with a set of tradeoffs—for example, the push and pull modes of the Observer pattern. Based on a selected pattern, the tool allows the designer to select a specific implementation tradeoff and generate the implementation code accordingly. The tool also supports configuring generation options that are applicable to all design patterns. The tool has distributed architecture and incorporates a hypertext rendition to give the users an online, web-based view reference to a pattern catalogue. The architecture of the tool was constructed to allow future extensions in terms of code-generation options and addition of new patterns to the collection.

With this tool, code can be automatically generated for a pattern by supplying application-specific information of a chosen pattern. The tool is specific for code generation. Code generation is an important aspect of a design pattern, since it adds to the utility and applicability of a pattern in application design. After all, it is the code that gets compiled and executed. Using this tool, designers can understand how design concepts that are captured in design patterns can be translated into code. It also helps in understanding how various tradeoffs are mapped into implementation.

This tool does not address the important issue of providing a design environment for integrating patterns. It works only on a per-pattern basis and does not provide support for interacting and merging pattern participants to develop application design. The tool can play a good role in the browsing activity of the POAD methodology but not in the integration and design processes. With the easy-to-use web-enabled interface, it is a useful tool for analysts and designers to browse a database of patterns and understand various implementation tradeoffs.

**The Pattern Lint**

The pattern lint tool [Sefika et al. 1996] checks the compliance of a pattern implementation to its design. The tool also has many other constraint-checking features that are not limited to patterns, such as checking concrete rules for coding styles, checking compliance to architecture styles, and checking heuristic guidelines like high cohesion and low coupling.

The pattern lint tool is used for evolvable systems where requirements change and new features are requested over time. In order to cope up with the new changes in a timely manner, application developers usually tend to make changes directly at the implementation level. Thus, the implementation usually diverts from the original design. This means that some design principles may be lost or violated at the code level, which motivates the need for consistency checks between the design and the implementation at all phases of the development lifecycle. Tools are required to assure that the implementation of the application abides to the design rules imposed at the application design level.

A design pattern provides a design solution that should be adhered to at the implementation level. When a pattern is used in an application design, the rules imposed by the design structure and dynamics modeled in the pattern design should be reflected in the code. The implementation of a design pattern should comply with the rules imposed by the pattern. The pattern lint tool is used to conduct such conformance checks.

To use the pattern lint tool, a set of rules is defined for each design patterns. The user selects the pattern and checks the implementation
against the rules associated with the selected pattern. The pattern lint tool can be used to check the conformance to both static and
dynamic rules imposed by a design pattern. Such integrated validation for static and dynamic aspects makes this tool superior to other
validation tools that check either the static or the dynamic aspects separately [Murphy et al. 1995].

As an example of the rules that can be checked using the pattern lint tool, consider the Mediator pattern [Gamma et al. 1995]. In this
pattern, the Mediator class has direct reference to all the Colleague classes, and all interactions between colleague objects should go
through the Mediator. The pattern lint tool can be used to check whether any of the Colleague classes has direct reference to another
Colleague. The tool can use both static and dynamic analysis to check for violation of such design rule. For instance, at the static diagram
level, the tool can check the implementation classes for direct references or programmatic declarations to a Colleague from within another
Colleague. From a dynamic perspective, the tool can execute the program to see whether a method invocation has occurred directly
between objects of type Colleague.

Apparently, this tool is not a pattern-based development rule. It is more useful for measuring conformance or validation of an application
implementation to the rules defined in each pattern. The tool does not implement a methodology to develop applications using patterns
and hence cannot be used for POAD development.

**Hooks and Templates**

A more global view of deploying patterns in design is proposed in by B. Pagel and M. Winter in “Towards Pattern-Based Tools” (1996).
The method is based on the premise that in order to support the systematic and controlled application of patterns in software design,
design tools should be able to treat all patterns like any other design primitive, such as classes, relationships, methods, and attributes.
Therefore, a metamodel for representing a pattern is needed. Such a metamodel would make the development tools open to use any of
the current or future design patterns.

Pagel and Winter propose using the Hook&Template pattern [Pree 1994]. The Hook&Template pattern defines which classes (methods) of
a pattern’s constituents are considered template class (methods). It also defines which classes (methods) are considered hook classes
(methods). Template classes and template methods are considered the frozen spots of a pattern—the spots that should be used as-is in all
pattern instantiations. Hook classes and hook methods are regarded as the hot spots of the pattern—those parts of the pattern in which
the designer will hook the application-specific details.

Based on the Hook&Template pattern, a pattern metamodel is derived. Every feature in the Hook&Template pattern is translated into a
class in the pattern metamodel. CASE tools that support OO designs can be extended according to this metapattern to be able to treat
design patterns as they treat other design constructs. Some of the metaclasses presented in this metamodel already have one-to-one
mapping with UML metamodel classes, such as the concepts of a class, a method, an association, or a parameter. Other metaclasses are
specific for the representation of a design pattern, such as the specialized methods HookMethod and TemplateMethod. Recall from our
discussion in Part 2 that the UML metamodel already covers most of the elements in the pattern metamodel on which the Hook&Template
approach is based.

This approach also emphasizes the pattern instantiation process, which refers to the process of applying patterns in concrete designs as
pattern instances by assigning roles defined in an abstract pattern to classes and methods of the concrete design. Whereas the original
patterns, as documented in the catalog of patterns, are called abstract patterns, the concrete application of a pattern in developing the
design of a specific application is called and instantiated pattern. Pagel and Winter define a process that uses a formal schema to
instantiate a pattern in the application design. They also emphasize the role of a pattern repository to ease the process of searching for
patterns and instantiating them.

To support the implementation of pattern-based tools, the Hook&Template pattern (metamodel) provides a unified description for all
patterns. This metamodel is used to distinguish between the pattern elements that are implemented by the user and those elements that
are already defined for the pattern class collaboration. This method is supported by a design tool, which helps the designer in determining
those aspects of the pattern that are the templates and those that are the hooks. Incorporating the concept of patterns and the related
design construct at the metamodel level is required in the POAD methodology, as we discussed in Part 2. It is unclear what the current
status is on the development of this tool. We believe that POAD models should be incorporated within the UML metamodel framework,
and hence tool support for UML modeling can be used for the POAD approach.
TogetherSoft Control Center [TogetherSoft 2002] is a comprehensive development environment that targets the end-to-end model-build-deploy goal. The development environment is based on UML for analysis and design with several backend programming editors for code and implementation. The environment has built-in support for design patterns and specifically for the GoF patterns.

In this development environment patterns can be used in two ways: using templates or using Java APIs. With templates, the code of pattern classes and methods is attached to each pattern in ASCII format. This code captures the fixed parts that need not be changed with specific applications. The code also provides placeholders for those parts that are application-specific. The user creates an instance of the pattern from the template, using the GUI, and provides the application-specific data to replace the placeholders, whether these placeholders are class names, link or reference relationships between patterns, or new classes to add to the pattern. At the end of the process, the UML model and the code are generated for the application-specific instance of the pattern.

The second method to use patterns in this development environment is to use a specific Java API provided with the tool. This API provides a unified interface that allows users to develop their own pattern and integrate it with the TogetherSoft development environment. The tool can then interrogate any attached pattern and present to the user what needs to be specified to instantiate an instance of this pattern.

The TogetherSoft modeling and development environment provides good support for design pattern instantiation through either the template or the API approaches. It also provides a sort of pattern repository that is extensible by submitting new patterns or creating your own. However, this environment has little support for the composition of patterns and solving the problems of pattern integration and interfaces. The tool is mainly for supporting analysis and design using UML with support for deployment and instantiation and creation of design patterns.

[Team LiB]
Requirements and Features of a POAD Tool

The POAD process for constructing designs by gluing together design patterns as components requires tool support to streamline the development process. The main concerns of many of the currently available tools are pattern instantiation, implementation, code generation, and application of a pattern to a specific design. The POAD approach is a visual process of designing applications by assembling patterns together as components and refactoring the design to produce dense, profound, and traceable design models. The development environment that we use for the POAD process should provide support for some features that pertain to the methodology. We discussed some of these features in chapters 7 through 10. In this section, we summarize some of these important features.

Patterns as First-Class Design Elements

A primary feature of pattern tool support is its treatment of patterns as first-class design elements. The development environment should treat a pattern as a design element in its own right and not as a derivative or an aggregation of any other design elements. It should provide the designer with a representation of the pattern in the design as a separate element that can be connected to other design elements. For example, using UML, a pattern can be represented as a stereotyped package that can be connected to other packages and other classes in the application design.

In POAD, patterns are represented as separate design components. The rationale about selecting a first-class representation of a pattern is that at some design level, it is enough to know that some pattern is used in the application, and it is not necessary to overwhelm the designer with the details of the internal design of the pattern or how it solves the application design problem. Patterns in POAD are treated as design building blocks from which applications can be developed. We believe that it is easier for the designer to work at a higher level than class diagrams but yet be assured that each of the elements used at that high level has well-proven class diagrams from which the design and implementation exist and that they are proven to have good design quality.

In POAD models, patterns are treated as UML stereotyped packages. An important concept for treating a pattern as a first-class design elements and a grouping mechanism is the use of interfaces. The development environment should provide support for pattern interfaces. Recall from previous discussions that a pattern can have multiple interfaces, so the development environment should provide support for selecting from multiple and alternative interfaces to give the designer the flexibility to use a pattern interface according to the application design context.

POAD Hierarchical Models

In earlier chapters we discuss the design models and diagrams required for the POAD methodology. The models used in POAD are based on UML notations in which stereotyped packages, interfaces, dependencies, and class diagram models are used. The development environment should provide support for modeling using those pattern diagrams and for the process of designing with patterns and documenting pattern relationships while hiding the pattern structure details not utilized at the high-level design.

The development environment should support the three logical views used in POAD: Pattern-Level view, Pattern-Level with Interfaces view, and Detailed Pattern-Level view. The diagrams developed under the Pattern-Level view are used to model the application as a visual composition of constructional design patterns. The diagrams developed under the Pattern-Level with Interfaces view are used to reveal and interconnect the patterns interfaces. The diagrams created under the Detailed Pattern-Level view are used to reveal the details of each pattern.

The three modeling levels used in POAD are hierarchical, and each design level hides the internal details in lower design levels. The designer should be able to traverse these hierarchical models to explore the design details or obtain a generalized view. Using an
integrated development environment that supports POAD, this activity should be supported in an automated fashion. For example, the
designer should be able to reveal and hide the class diagram model of each pattern. At the Pattern-Level view, the designer sees the
patterns as black boxes; at the next level, he sees the pattern as boxes with interfaces; and at the detailed level, he sees the internal
design of each pattern using a class diagram. Traversal down and up these hierarchical models should be supported in the development
environment.

Traceability

An important issue in using patterns to develop software applications is that by the time the application design is created, the pattern
essence is lost [Soukup 1995]. We are no longer able to identify the pattern in the design. This is because the application design is now
presented using class diagrams, which capture no information about which pattern is used and where. Many pattern-oriented design
approaches suffer from this problem because the pattern is not preserved in the design as a first-class element or it was used as a
design element at some stage but nothing remains of the pattern except the class model, which is now dissolved in the rest of the
application design.

POAD avoids this problem by using patterns as design elements and capturing all the models generated from the time patterns are used
as black boxes to the time they are dissolved in the application design. However, capturing the models is not sufficient. The exercise of
tracing element across those models is too difficult to do manually. The development environment used for POAD should provide support
for such traceability mechanisms.

Traceability is a modeling property that supports uninterruptible transition from high-level abstractions to lower levels. The effort in tracing
high-level abstractions to lower levels should be minimized both semantically and visually. There are several ways the design
environment can help achieve traceability in the POAD models:

- **Capture refinement activities.** During the design process, the designer works at various levels of abstraction, from high-level
diagrams such as the Pattern-Level views to lower-level diagrams such as the Detailed Pattern-Level views. The decisions
made at the design stage are mostly refinement decisions, which should be captured by the development tool and saved in
the underlying model. The tool should provide means to trace between elements from coarse-gained to fine-grained views.
For example, pattern dependencies in Pattern-Level view are traceable to class/class, class/operations, and
operation/operation relationships in the Pattern-Level with Interfaces view. This traceability feature enables the designer to
find out how elements are refined at lower design stages or to find out the course-gained representation of the element at a
higher design level.

- **Tracing interfaces to implementations.** The interfaces provided by the pattern are implemented by its internal classes. The
designer should be able to trace which design element inside the pattern implements the pattern interface. In an IDE
environment, this could be made possible by providing the designer at the Pattern-Level with Interface with the option to trace
down the implementation of the interface at the Detailed Pattern-Level diagram. At the Detailed Pattern-Level diagram, the
tool could also use a connectivity mechanism to show the connection between the interfaces and the pattern internals. This
connectivity mechanism could be as simple as a UML realization relationship between the interface and the classes in the
internal class diagram model.

- **Top-down traceability.** In POAD the designer starts with the Pattern-Level diagram, which is a representation of the design
using patterns only, and ends up with the application design, which is the class diagram model. Starting from the high-level
view of the design as a composition of patterns, the designer might want to find out which classes in the application design
implement the patterns after all the design iteration and refinement activities conducted during the POAD design process. A
traceability mechanism allows the designer to traverse the design models down the design abstraction hierarchy. This
traceability mechanism is a function that should be supported by a POAD development environment. The designer should be
able to select a pattern at the pattern level and use a tool to automatically trace this pattern all the way to the application
design. The tool would present to the designer the history of how the pattern evolved and the final classes in the application
design that are now used to implement this pattern.

- **Bottom-up traceability.** During the design-refinement process, participants from several patterns are merged to develop the
application design. As a result, a class in the application class diagram could play several roles from different patterns. The
designer might want to trace a particular class in the application class diagram back to the one or more patterns in which it
plays a role. A POAD development environment should provide a traceability mechanism by which the designer can select a
pattern in the application class diagram and use a tool to automatically trace this class all the way up the design abstraction
levels to the patterns in the Pattern-Level diagram. The tool would also present to the designer the history of how the pattern
participants were merged or grouped to create this application class.
Pattern Repository

The POAD methodology is based on using patterns as components. We look for design patterns to cover as much of the application design as possible, leaving little to the creation of new design elements that are not patterns. To achieve this goal, there should be a repository of design patterns from which the designer can select patterns for the application under development. After all, we cannot build an application using design components when we do not have these components.

We can identify several collections of patterns (such as the GoF patterns or the PLoPD series patterns). Some of those collections are distributed as standalone packages or are bundled in the distribution of a development or modeling environment (such as TogetherSoft Control Center [TogetherSoft 2002]). However, we cannot find a library that has the huge collection of patterns that have been produced over the years. Such a pattern library should be kept in a persistent location or database and should provide easy access to the content. The indexing of existing design patterns, developed by Linda Rising [Rising 2000], is a significant start toward achieving the ultimate goal of a pattern database. Until such a database becomes a reality, the analyst will have to use propriety pattern collections as inputs to the POAD process.

The development environment used in POAD should provide support for connecting to a database of patterns. Support for browsing and querying the database is also required. To reuse a design pattern as a building component, the storage of the pattern in the database should facilitate the retrieval process. The repository should keep at least the following information about the patterns:

- The traditional pattern documentation, including the context, problem, forces, consequences, sample code, and so on. Several templates, such as the GoF or POSA templates, can be used.
- The pattern interfaces. There can be multiple interfaces for a pattern. All interfaces should be stored with the pattern models.
- The pattern model, which holds the structure of the pattern that can be embedded into the application design. In POAD we are mostly interested in the class diagram models.

Models with Memory

During the design-refinement process in POAD, there are several activities in which the names of the model elements are changed. For example, in the instantiation activity of a pattern, the application designer specializes and concretizes the generic elements in the pattern model and converts them to application-specific instances.

Tracking the history of changes made to the pattern is an important feature for the POAD methodology. It is this history-tracking feature that enables traceability between various design models. Therefore, the development environment should provide support for saving the changes done by the designer during the instantiation of a pattern. For example, if we assume that a system without memory is used, we have no way of linking application-specific classes or methods to their original names in the pattern class diagram model. In many design methodologies that use patterns, this link is kept in the designer's mind and is not captured in the design models. In POAD we explicitly emphasize this feature. A development environment supporting POAD should implement this feature.

Another activity in the POAD methodology that requires models with memory is the merging and grouping activities during the design optimization process. In these activities two design classes could be merged into one application class, and if they are, their names are changed. The environment supporting the POAD methodology should provide this merge feature. The designer should be able to select two classes for merger and use a tool to generate the resulting class. The merge could result in naming conflicts between the design elements within the merged classes. The tool should provide a means to present such conflict to the designer and a means for renaming. After the merge process, the model should keep track of which original two classes are not represented by the new class and how the new class is traced back to the original participants at higher abstraction levels.

Pattern instantiation
Pattern instantiation is the first activity in the design refinement phase. During this activity, the designer transforms the abstract and generic design in the design pattern model into an application-specific instance suitable for use in the application under development.

The POAD development environment should support the pattern instantiation process. Some of the tools that we discuss in the previous section support pattern instantiation using a wizard that walks the designer through the instantiation process. The fixed spots of the pattern's design (also called cold spots) are usually kept without changes. The variable spots of the pattern's design (also called hot spots) are presented to the designer to fill in with application-specific details. For example, the instantiation wizard could ask the designer for the names of the concrete strategy classes to create for the instantiation of a Strategy pattern. Similarly, other hot spots, such as methods or association parameters, could be presented to the designer for instantiation.

During the design-refinement process, the complexity of the design is reduced by eliminating replicated abstract classes or interfaces that result from using several application instances of the same pattern type. The development environment used in POAD could be used to find out these replicated abstract classes. The tool could track down the pattern-instantiation process and prompt the designer for possible replicated classes that could be merged.
Summary

In this chapter we discuss the tool support for designing applications using design patterns. We selected some of the existing tools that support design patterns at various development phases, from modeling to implementation.

We can conclude from the above discussion that the tools currently supporting design patterns emphasize pattern instantiation, documentation and browsing, cataloging, code generation, and implementation. While all those aspects of a design pattern are important for the POAD process, there is still one important aspect that is lightly addressed by those tools: composition and integration of patterns with each other and with other parts of the design that are not patterns. The tool support for composition of patterns at high design levels is essential to the success of the POAD methodology. Such composition tools and development environments should provide support for the hierarchical views of POAD models and traceability from one modeling level to the other.
Chapter 17. Wrapping Up

Design patterns cannot possibly become part of streamlined software development process unless we develop practical methodologies that essentially use patterns in their core development process. Pattern-based development processes in turn cannot be successful and widely adopted unless they solve the issues related to composition of patterns.

POAD is about designing software systems by composing design patterns in a systematic manner. We have discussed three aspects of the methodology:

1. Technological aspects, including the visual design models,
2. Process aspects, including the steps to use POAD, and
3. Usability aspects, including application of POAD in several case studies.
Systematic Composition of Design Pattern

Whereas design patterns are touted as means of developing OO designs, the development process lacks systematic support to utilize these proven design solutions. We can generally classify approaches to designing with patterns as *ad hoc* and *systematic*:

1. **Ad hoc.** Much of the current work regarding pattern use falls into this category. A design pattern captures the solution and the process of applying this solution. A systematic approach to design with patterns goes beyond just applying a certain pattern. For instance, using a Strategy pattern to design control applications or using an Acceptor or State pattern to design communication establishment in distributed applications are not considered systematic utilization of patterns. This is because there is no process to guide the development, and the development steps are not repeatable.

2. **Systematic.** We identify two categories under systematic development with patterns:
   
   a. **Pattern languages.** A pattern language provides a set of patterns that solve problems in a specific domain. Pattern languages capture not only the patterns themselves but also the relationship between these patterns. They imply the process of applying the language to completely solve a specific set of design problems.

   b. **Development processes using general-purpose patterns.** A systematic development process defines the analysis and design steps, models to support development, tools to automate the application of the development steps, and a repeatable method.

The POAD methodology is concerned with the latter: the systematic development processes using design patterns. Systematic development using patterns embodies a composition mechanism to glue patterns together at the design level. Composition techniques can be classified as *behavioral* and *structural* composition, as discussed in Chapter 2. POAD advocates a structural approach to compose constructional design patterns.

*Part I* includes a summary of pattern composition approaches that are currently known and widely applied. We classified these approaches as behavioral and structural composition approaches. Some of these approaches are useful attempts to develop systematic composition methodologies of design patterns. The discussion in *Part I* summarizes each pattern composition technique and the advantages and limitations of each technique. We also included a comprehensive further reading section.
POAD Characteristics

The POAD methodology is unique in integrating the stringing and overlapping patterns approaches discussed by Alexander in *A Pattern Language* (1977). While the stringing patterns approach is easy to start with, it leads to non-dense design. While the overlapping pattern approaches lead to a dense design, they are difficult to deploy early at the design level. POAD reaps the benefits of the two worlds. It provides the ease of stringing patterns at early design phases and the ability to produce dense design by overlapping patterns at later design phases.

POAD is applicable early at the analysis and design phases of software systems. As opposed to several techniques that use design patterns at the code or detailed design level, this methodology works from the top level down: it can be used to develop high-level models that convey few details and then to proceed to lower-level design details.

POAD provides a new perspective on developing software using design patterns as building blocks. POAD uses the pattern's class diagram models. This leads to a class diagram for the system, which can be more easily understood and implemented by the developers than can other behavioral models, which require further translation to implementation models.

Traditional OO development methodologies create application designs that are represented using a class diagram. POAD produces application designs using composition of constructional design patterns. Representing the design as a composition of patterns is useful for documentation, maintenance, and reuse. Moreover, using well-established design patterns as components can result in good design quality.

POAD, as a methodology, has some important characteristics; it is

1. **Consistent.** The inputs and outcomes of each step are clearly defined such that if the process is performed again and again, it will consume the same types of inputs and produce the same types of outputs. The inputs and the deliverables of each step are discussed in Part III.

2. **Manageable.** Manageability of the process ensures that the technique is practically feasible. A tool support for the process helps in managing the development steps.

3. **Traceable.** The process ensures that the outcomes of each step are used by and traced into the next step. Traditional OO design methodologies produce models that are traceable from one step of the development phase to another (such as from analysis to design). On the contrary, structured analysis and design methodologies have a huge gap between the models produced during the analysis and design phases. POAD adds more traceability to the traditional OO techniques, since it is traceable from the pattern-level models (high-level models) to the class-level models, as discussed in Part III.

4. **Composable.** The process utilizes design components that are reusable across several applications. POAD defines how these components are composed together to build application designs, as discussed in Part II.

5. **Automatable.** As discussed in Chapter 16, tool support for the development process is feasible. The tool facilitates the development of application designs and integrates with commercially available OO modeling tools.
POAD and Software Reuse

Software reuse can be defined as "the process whereby an organization defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in development activities" [Mili et al. 2001]. For a long time, reusable artifacts have been thought of as source code or black-box binary components. Though the benefit of reuse at this level (code or binary) is maximal, it is unlikely to happen. This is because the component is usually developed for a specific context, and it may be hard to modify the code due to some constraints, including consistency, validation, and contractual constraints.

We believe that the realistic and practical level of reuse is the design level. Ideas about how to develop a design to implement some common structure or behavior are more likely to be reused than are source code or binary components. In promoting the idea of reuse at the design level, we have developed the POAD methodology.

The POAD methodology leverages reuse to the design level in a systematic fashion. A major challenge in developing software is the severity of the early design decisions. At the design level, we do not have sufficient knowledge to ensure that our design decisions are good. Reusing good designs can help designers address this challenge. POAD is built on the premise that we can build applications by composing those reusable design components. Using design patterns, we decrease the risks of making bad decisions, since we reuse ideas and designs that have proven useful and applicable in other systems.

POAD is a reuse-based approach to software development. It reduces the development time and effort by reusing previously documented, good-quality designs. It also carries with it all of the challenges of a software reuse methodology, which include retrieval and selection of patterns, integration and composition, adaptation, management, organizations, and so on [Mili et al. 2001].

POAD is a complete software reuse approach. It defines the logical views and models to be developed, model constraints, the development process, and the tool support requirements. It is highly based on using components at the design level and is an interface-centric development methodology.
Applying POAD

This book provides several case studies that illustrate how to apply POAD as a pattern composition methodology. They range from simple case studies (chapters 11 and 12) that get the reader started in a short time to complex case studies (chapters 13 and 14) that illustrate how the methodology is applicable in large-scale industrial applications.

Part II and Part III describe the technological aspects and the process aspects of the POAD methodology respectively. We discuss Part IV four case studies, each in a separate chapter. Each chapter illustrates the application of the POAD steps to develop the design models for a particular application. The first two case studies are small-scale examples; each is constructed with just a few constructional design patterns. These two chapters are used to speed the learning process of applying the POAD methodology. The first case study is the design of a feedback design framework that is usually used in reactive systems and in the quality-control phase of a production line. The second case study is the design of a framework for simulating the behavior of waiting queues, which has evolved from a software reuse experiment conducted in the classroom environment. Addy et al. 1999.

The other two case studies are large-scale applications. The first case study explains the application of POAD to an industrial-scale application in the digital content remastering domain. The second case study illustrates the application of the methodology to part of a large-scale distributed medical informatics system. The diversity in the nature of the applications developed by POAD illustrates that there is a wide range of applications that could be designed using the POAD methodology.

The simple case studies are of great help in bootstrapping a novice user to apply and compose patterns. The detailed process as well as the simple nature of the applications illustrated in the book provide comprehensive examples of applying POAD. We also reference other pattern composition approaches and initiatives for expert users to compare and analyze.

We advise you to experiment with the two case studies in chapters 11 and 12 before applying the POAD methodology in developing your own applications. We also recommend a comprehensive study of the procedures outlines in Part III. Some of these steps provide only guidelines without enforcing specific techniques; the designer will have to select and define the techniques to use at these steps. For instance, in the acquaintance step, some designers might prefer online browsing of patterns, some might prefer reading from the PLoD references (PLoPD4, PLoPD3, PLoPD2, and PLoPD), and others might prefer the index provided in The Pattern Almanac. Rising 2000.
Pattern Literature

Though the field of design patterns is fairly new to the software engineering paradigm, books about design patterns, including cataloging, applying, deploying, and modeling design patterns, are plentiful. Following is a list of books related to the application of patterns in software development:

1. **A Pattern Language**, C. Alexander, S. Inshikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-king, and S. Angel (1979) and **The Timeless Way of Building**, C. Alexander (1977), Oxford University Press, New York. Much of the work in software design patterns is inspired by the ideas that Christopher Alexander and colleagues developed in the field of civil engineering several decades ago. These two books have been a great deal of help to authors in the software pattern community. The idea behind POAD is inspired by many of the statements in these two books, specifically the ideas of composing pattern by stringing and overlapping them.

2. Collection of books on pattern cataloging. There are plenty of books that capture general-purpose and application-specific patterns. Examples include catalogs of patterns for software architecture in **Pattern-Oriented Software Architecture: A Pattern System**, F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996); the yearly documentation of general-purpose and application-specific patterns in the **Pattern Language of Program Design** books PLoPD1, PLoPD2, PLoPD3, PLoPD4; the general-purpose set of patterns by the Gang of Four, **Design Patterns: Elements of Object-Oriented Software**, E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995); and **the Pattern Almanac**, Linda Rising (2000).

3. **Objects, Components, and Frameworks with UML: The Catalysis Approach**, Desmond D’Souza and A. Wills (1999), Addison-Wesley, Boston. D’Souza and Wills define a component-based approach to developing software that is heavily based on interfaces at both the design and implementation levels. The approach is called Catalysis, which is used to build object- and component-based systems using UML and some extensions made to UML constructs. At the design phases D’Souza advocates using frameworks as building components where a framework is "a pattern of model or code that can be applied to different problems" and OO frameworks are "collaborations with a default, skeletal implementation.” POAD takes a similar approach to designing software as compositions of constructional design patterns; however, we distinguish patterns and frameworks. The Catalysis approach addresses many issues in developing software, such as composing physical components, distribution of components, business-driven solutions, and so on. POAD addresses a specific problem: composition of patterns to develop robust software designs.

4. **Analysis Patterns**, Martin Fowler (1997), Addison-Wesley, Boston. Fowler discusses recurring patterns that analysts use in analyzing systems. This is another catalog of patterns but specifically for frequently recurring techniques used to analyze systems.

5. **Working with Objects: The OOram Software Engineering Method**, T. Reenskaug (1996). Reenskaug developed the Object Oriented Role Analysis and Software Synthesis method (OORASS, later called OOram), which uses a role model to abstract the traditional object (behavioral) model. The notion of roles focuses on the responsibilities of an object within the overall group of objects.

6. **Composite Design Patterns**, Dirk Riehle (October 1997), proceedings of OOPSLA’97, pp. 218–228, Atlanta, Georgia. Riehle uses role diagrams for pattern composition in this work. OOram is a behavioral composition approach; POAD is a structural composition approach. Behavioral pattern composition approaches have several limitations, as discussed in the Part I.

7. **Design Patterns and Contracts**, J. Jezequel, M. Train, and C. Mingins (1999), Addison-Wesley, Boston. Contracts are used to specify the behavior of design patterns. Jezequel and colleagues show how this could help in composing design patterns based on the pattern's behavior.

8. **Pattern Hatching: Design Patterns Applied**, J. Vlissides (1998), Addison-Wesley, Boston. This book does excellent job of showing software designers how to apply a pattern, instantiate patterns in designing applications, choose from several design tradeoffs, and understand the consequences of applying a particular pattern. The book is example-driven and shows how several patterns are put together to develop a design. What is missing in this book and is addressed in POAD are the systematic application of patterns at high-level design, definition of models for pattern composition, traceability mechanisms that assure classes of specific patterns are not lost, and procedures to develop these models.
Though this list is not comprehensive, it illustrates the diversity of approaches for using patterns in software development.
Future Trends in Pattern Composition

In the process of developing and applying the POAD methodology in several case studies, we found that there are several opportunities for improving the methodology and promoting pattern composition approaches. These improvements include:

*Integrating behavioral and structural composition mechanisms.* Patterns can be glued together at the design level using structural or behavioral composition approaches, as discussed in Chapter 2. The POAD methodology is a contribution to the structural composition mechanisms. We believe that the dynamic behavioral aspects of a pattern are as important as its structural aspects. Therefore, a concurrent structural (static) and behavioral (dynamic) composition analysis and design could produce better designs. One way of doing this is to define global scenarios between design components and local scenarios inside each individual pattern. These global and local scenarios can be integrated to develop analysis scenarios for the application and guide the process of developing the optimized class diagram.

*Cataloging patterns.* POAD is a reuse-based software development process. Any software reuse process is heavily dependent on a database or catalog of reusable artifacts. In the POAD methodology these reusable artifacts are patterns; therefore, POAD relies on a comprehensive database of constructional design patterns. A storage structure with some defined template is required. Since different patterns have different formats, this storage template may not be applicable to all patterns. The storage structure affects the retrieval and browsing criteria of patterns and hence affects the development process. An investigation is required to define an appropriate storage structure and the retrieval and browsing criteria as applied to constructional design patterns.

This research is related to the traditional reuse library issues. We envision that the research on retrieval and storage of patterns should be based on an analytical approach that defines what a pattern can possibly abstract.

*Architecture Issues.* In Chapter 4 we discuss the relationship between pattern-oriented designs and software architecture research. We believe that experimenting with architecture description languages and architecture styles as applied to pattern-oriented designs could further formalize the structural construction of pattern-oriented architectures. We also believe that the POAD methodology is suitable for constructing product-line architectures for a family of products. This is because the reference architecture of a family can be constructed as a composition of patterns. Each pattern provides an abstraction of a structure as a set of collaborating classes with enough flexibility to accommodate variations in product-specific details.

*Requirements Analysis.* One of the main difficulties in using POAD is analyzing requirements for the purpose of identifying which patterns can satisfy the required functionality. In many cases (especially in reactive systems) the requirements are specified as state machines or statecharts, and hence the state machine and statecharts patterns can be the direct solution. However, this is not the norm. An investigation is required to identify how constructional design patterns can be recognized based on user requirements. We illustrated in our discussion that traditional OO models, such as use case modeling and interaction diagrams, can be used to analyze the application and discover the required functionalities. However, applying use case-driven approaches to pattern selection is a topic for further investigation.

*Tool Support.* As discussed in Chapter 16 and in H. Xue's *Tool Support for Pattern-Oriented Design: GUI, Browser, and Database components* (1999), tool support for the POAD technique is inevitable. Current tools may support the POAD models syntactically, but they lack several important functionalities, such as the traceability and history features.

*Maintainability Analysis.* Documenting the design using pattern-level views can help in comprehending the application design as a composition of building blocks that are larger in granularity and richer in design information than are classes. Looking at a design as a synthesis of patterns can improve the quality of documentation, since design tradeoffs are documented in a pattern together with the consequences and forces driving the use of each pattern. We envision that the POAD methodology improves the documentation and the maintainability of application designs because it provides models of interfacing patterns that are traceable to classes in the final
class diagram of the application, and vice versa. Further research and studies are required to study the effect of the POAD models on maintenance effort and time.
Conclusion

We have found the POAD methodology useful in designing several software applications as we discussed in Part IV. We hope you will find the approach useful as a systematic way of composing patterns and applying them to the software development process. We expect in the future that several additions to and variances of the POAD methodology will appear, as we believe that a methodology is a living thing that gets its liveliness from continuous application to real-world problems.
Appendix A. Interfaces for Constructional Design Patterns

The POAD development process glues constructional design patterns. Interfaces qualify a constructional design pattern to be a unit for composition. In this appendix, we define interfaces for some of the patterns that we used in developing the applications in Part 4. There is no unique definition for interfaces of a specific pattern; a pattern can possess multiple interfaces.
Strategy Pattern

The Strategy pattern has the class `Context` as the interface to the encapsulated control strategy. One might also use the `ContextInterface()` operation as an interface operation for the Strategy pattern.
Observer Pattern

The Observer has two interfaces that allow coordinating the subject observed with its observers. These interfaces are the `notify()` interface operation in the subject and the `update()` interface operation in the observer.
Composite Pattern

The interface of a Composite pattern is the Component class by which its constituting elements, whether simple (leaf child) or composite (parent), interface to other components.
Reactor Pattern

The Reactor pattern has two types of interfaces, one for delegating the processing of events to other classes—the EventHandler interface class—and the other for receiving and scheduling events—the Dispatch() interface operation.
Template Method Pattern

The interface of a Template Method pattern is the `TemplateMethod()` operation of the `AbstractClass` class, which contains the skeleton of the applying step of an algorithm; each step is implemented by concrete subclasses.
Proxy Pattern

The Proxy pattern has one interface, the Subject interface class, which defines the common interface for the real subject (the remote machine) and the proxy (the local representative).
Abstractfactory Pattern

The AbstractFactory pattern has two interfaces. The first interface is for the factory object AbstractFactory interface class, which provides an interface for creating various types of products. The second interface is for the product created by the factory, the AbstractProduct interface class, which provides an interface for all various products created by the factory.
Builder Pattern

The Builder pattern has the Director as an interface, which builds the product.
Mediator Pattern

The Mediator pattern has interface as a set of Colleagues for which the mediation of messages is required.
Command Pattern

The Command pattern has the Invoker and Receiver as interface classes.
Appendix B. State of the Art and Practice in Design Patterns

This appendix provides a synopsis of the state of the art and practice in design patterns. The body of literature in design patterns has significantly grown over the years. Pattern mining, pattern specification, tool support for pattern-based development, pattern-based implementation, and generic programming are important areas where significant achievements are noticed. The following discussion elaborates on the achievements in these areas. We then discuss several unresolved problems and issues where more work is still needed.
Pattern Mining

Many patterns are currently documented in the literature. Searching for new patterns, or so-called mining [Buschmann et al. 1996], is an ongoing activity in the pattern community. There is no clear classification of the discovered patterns. Some patterns are related to technical issues like architectures, design, and implementation patterns. Other patterns are organizational and process patterns. Domain-specific or application-specific are other classifications. An activity that is associated with pattern mining is cataloging and organizing the discovered patterns (indexing patterns) [Rising 2000]. Every experienced designer works on documenting and mining useful patterns in her or his field of expertise. The yearly literature [PLoPD, PLoPD2, PLoPD3, PLoPD4] is the output of the mining activity.
Pattern Description Formats (Templates)

Several pattern description formats are currently used in pattern documentation. The basic Alexandrian [Alexander et al. 1977] pattern format has the context-problem-solution sections. The GoF [Gamma et al. 1995] pattern format has more detailed sections on the solution structure and participant. The POSA book [Buschmann et al. 1996] uses another format in which more emphasis is placed on the driving forces to use the pattern and the resulting context and consequences. There is no clear distinction on when a format is best suited to a specific type of patterns. Choosing the adequate format is essential to convey the pattern information to the pattern user and hence increase the understandability of a pattern as a design-building unit.
Tool Support

Some work has been done on tool support for pattern usage and documentation. The objective is to provide CASE tool support for patterns and automate their usage. Many experienced system designers argue against the idea of using automated tools for patterns. They argue that patterns are mental building blocks that are more related to human understanding than to automatic usage. Using patterns in a design is not just a simple process of interconnecting the components of the patterns together. We have to merge some components, which may lead to some problems, such as losing track of the pattern or increasing the complexity of the resulting application.
Pattern Formalism

To formally specify a pattern, we can specify

- The behavior of the constituting elements, such as the objects and classes.
- The behavior of the overall pattern as a design component.

The work on formally specifying design patterns lacks the component specification nature and is more concerned with specifying the individual constituents of a pattern. The following initiatives address the pattern specification issues:

- A formal approach to design pattern definition and application is proposed by Alencar and colleagues in "Approach to Design Pattern Definition & Application" (1995). Alencar uses the Abstract Data Views and Abstract Data Object ADV/ADO [Cowan & Lucena 1995] to describe schema for classes of a pattern. The pattern has a different schema other than that used for its classes. The pattern schema is called a pattern constructor and encompasses the ADVs, ADOs, and their relationships.

- Eden, Gil, and Yehudai proposed a formal language for design pattern called LePUS [Eden et al. 1998] for precise specification and automatic application of design patterns [Eden et al. 1996; Eden et al. 1997]. The LePUS language approach formally specifies the classes used in the pattern and their relationships, but it does not present the pattern as a design component. The LePUS graphical notation focuses on the inside of the pattern rather than on the pattern interaction with other design artifacts.

- Jan Bosch proposes that design patterns should be part of the programming language itself; that is, design patterns should be language constructs [Bosch 1998b]. Bosch proposes the Layered Object Model (LayOM) as a language with explicit support for representation of design patterns. The LayOM approach adds two layers to the usual class definition, one for the structural aspect and the other for the behavioral aspect. The layers interact with each other through interfaces. However, the equivalent C++ code for such a case is one class for the whole pattern, and thus both the programmatic and the structural design of the pattern are lost. Also, it is not explained how this approach aids gluing patterns together in applications; it focuses on representing individual patterns only.

- Mikkonen describes an approach to formal specification of design patterns [Mikkonen 1998]. He shows that a rigorous reasoning can be eased by formalizing temporal behaviors of patterns in terms of high-level abstraction of communication.

Other approaches to formalize design patterns can be found in Khriss and Keller's "Transformations for Pattern-based Forward-Engineering" (1999). The above approaches to formalizing the representation as well as the specification of a pattern ignore the interface aspect of a pattern, which is the main qualifier for gluing it with other design patterns and hence qualifying the pattern as a design component.

Note that studies on formal specifications of design patterns have not impacted the design pattern community and software design practitioners. The reason experts give for this lack of interest is that the formalisms used to specify patterns are often large and complex, which makes them hard to understand and use. This defeats many of the benefits of patterns, which are most effective when they enhance rather than impede communication among software developers.
Pattern Languages

Pattern languages have evolved as an approach for documenting a set of closely related patterns. They provide a vocabulary for using patterns for a group of related design problems. They also define a process for resolving software development problems systematically. The well-known series of books on Pattern Language of Program Design (PLoPD) contain many examples of such pattern languages. As an example of how a complete pattern language can be used to solve a set of related design problems, we presented in the fourth volume of the PLoPD series a pattern language for finite state machines [Yacoub & Ammar 1999]. The language consists of design patterns used for producing an object-oriented design for software systems specified using finite state machines. Schmidt and colleagues, in Pattern-Oriented Software Architecture (2000), illustrate another example of a pattern language for developing distributed-object computing middleware, applications, and services. Pattern languages are gaining more popularity because they illustrate how several patterns can be used together to solve a larger problem rather than the individual problem-solution focus of an individual design pattern.
Pattern-Based Implementation

A recent interesting book by Andrei Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied (2001), addresses the issue of closing the gap between pattern-based design and implementation, where pattern features might change at the implementation level. The book describes a library of generic components based on C++ templates, which implement known design patterns such as singletons, visitors, proxies, abstract factories, and others. This library can be used in many applications due to the concepts of generic programming. As John Vlissides described the components presented in this book in his forward message, that these "generic components raise the level of abstraction high enough to make C++ begin to look and feel like a design specification language."

Java-based implementations of patterns for concurrent programming applications have also been addressed by Lea in Concurrent Programming in Java: Design Principals and Patterns (2000). Patterns are used for defining concurrency-based Java threads and for synchronizations mechanisms.
More Challenges

Though a lot of work has been expended in advancing the state of the art and practice of design patterns, there are still many challenging problems to solve before design patterns become a mainstream practice in software development.

Formal Specification of Design Patterns

Some researchers find that lack of formalism in the world of design patterns impairs the capabilities of pattern description, and hence automation of pattern usage might not be possible. The problems in pattern formalism include questions such as How much of the pattern information can be scientifically specified? What attributes are we trying to assure by using a formal language? Will the formal definition lessen the time to implementation and code generation? Can the formal definition help in validating the design decisions made in the pattern design? Many practitioners do not find a direct benefit from having a well-defined formal specification of a pattern, since the specifications do not save them implementation and development time.

Pattern Instantiation and Implementation

Gluing patterns in a design is not just a simple process of interconnecting the components of the patterns. We have to merge some components that may lead to losing track of patterns or increasing the complexity of the resulting application. Several issues arise when implementing a design pattern:

- **Traceability**: Programming languages don’t support a language construct for representing and coding a pattern. When a pattern is instantiated in a design, in several cases its constituents will be used for other purposes. In the final application design, the original constituents of the patterns might not be easily traceable. Soukup pointed out that when implementing design patterns, programmers create, extend, and modify classes throughout the design. This creates a major maintenance and traceability problem as programmers “tend to lose sight of the original patterns.” [Soukup 1995; Hedin 1997; Algerbo & Cornils 1998; Bosch 1998b].

- **Reusability**: For design patterns, the level of reusability is design. Since patterns have no explicit implementation support in languages, the implementations of design patterns are not eligible for reuse; they are mainly considered as guidelines and samples of implemented designs.

- **Implementation Overhead**: The overhead in implementing a design pattern is that the developer often has to implement several methods with trivial behavior, such as delegating a message to another object. This leads to significant overhead for the developer and decreases the understandability of the produced code.

Development Methodologies

Patterns have been successfully applied in several applications, such as telecommunications systems [Schmidt 1995] and distributed applications [Schmidt et al. 2000]. They play an important role in application frameworks such as ET++[Weinand et al. 1988]. But there is
no clear technique or methodology for developing applications and frameworks using patterns. Until now, the instances of applying patterns in framework development are coincidental; that is, there is little concern about how to initially design the framework based on patterns and to further proceed into detailed design in terms of classes.

**Documentation of Design Patterns**

Patterns exist in almost every field in software systems. Patterns can belong to a specific domain or a specific application. They can be organizational or process as well as analysis, design, and implementation patterns. Cataloging of patterns in databases still lags behind the celerity of producing them. Classification as well as database documentation of patterns is quite essential [Rising 2000].

**Evaluating Existing Design Patterns**

There are currently no guidelines on how to evaluate the quality of a design pattern. The ability to utilize a pattern in several practical applications is the only available qualitative measure. Recently, several research projects have been initiated to evaluate the quality of design patterns themselves [Algerbo & Cornils 1998; Gil & Lorenz 1997].
Glossary

Many terms in software engineering are overloaded with a number of meanings. For instance, a software architecture has different meanings to different people. Architecture can mean the high-level view of a system in terms of components and connectors. It can also mean the rules governing how the system should be built. This appendix describes the terms as we have used them in this book.

Analysis

The process of identifying important concepts such as classes in object-oriented analysis or patterns in pattern-oriented analysis. Analysis is a separation of a whole into its components: an examination of a complex whole, its elements, and their relations.

Architecture

A description of a system that is defined by (1) structure aspects, which define the system decomposition into a related set of components and the relationships between components (connectors and interfaces); (2) functional aspects, the responsibilities and functionalities allocated to each component; and (3) behavioral aspects, the interaction among components in a specific processing situation or execution scenario.

Bottom-up software development

An approach to software development in which progress is made by composition of available components.

Class associations

Relationships between classes, which can be aggregation, composition, generalization, and dependency, as specified in UML.

Class library

A set of classes that can be used in the solution to a particular problem. Although these classes are sometimes abstract and sometimes have relationships, they do not define an abstract structure for a system, and there is no inversion of control: the user writes some custom code that calls the class library code when needed.

Collaboration
A collaboration between two objects indicates that these objects will send or receive messages in the course of satisfying responsibilities.

**Component**

The fundamental unit of large-scale software construction. Every component has interfaces and internals. Components have different natures; they can be design components, executable binary components, or code components.

**Constructional design pattern**

A design pattern that provides a solution as an abstraction of a common design structure in the form of a UML class model. Constructional design patterns are design components that can be glued together at a high design level. Constructional design patterns are used in constructing the structure of the design using their class models. The internal details of the constructional design pattern are hidden at high design levels (pattern views) and are traceable to lower design levels (class views). A constructional design pattern is a design pattern, it has interfaces, and it has a class model; that is, it provides a solution as a structure of collaborating classes.

**Dependency**

In general, a dependency implies that the complete functioning of an element requires the presence of another, which exists in the same level of abstraction or realization (i.e., pattern, class, or framework level of abstraction).

**Design**

The process of converting analysis models to detailed responsibilities and role models within the boundaries of the software system.

**Detailed pattern view**

A schematic diagram that refines the pattern-level with interface view by tracing the pattern interface to the internals of a pattern.

**DICOM**

Digital Image and Communication in Medicine, a standard for communication in medicine.

**Domain analysis**
1. The analysis of systems within a domain to discover commonalities and differences. 2. The process by which information used in developing software systems is identified, captured, and organized so that it can be reused to create new systems within a domain.

**Domain**

A distinct functional area that can be supported by a class of software systems with similar requirements and capabilities.

**Method**

An approach to activities generally adhering to common principles. A methodology goes beyond this approach to include management, testing, quality, reuse, and architectural design.

**Methodology**

A systematic collection of techniques and guidelines for building, buying, maintaining, and enhancing software production. A methodology provides a basis for communication, techniques, and reusable software engineering discipline. It is a collection of methods applied across the software development life cycle and unified by some general philosophical approach.

**Modularity**

The property of the system that has been decomposed into a set of cohesive and loosely coupled modules.

**Pattern dependency**

Relationships between patterns in a pattern-level view. A pattern dependency indicates a semantic relationship between two patterns, a situation in which a change to the source pattern may require a change to the target pattern, or a pattern delegates the processing of a certain function to the other. This relationship is further refined in later design phases to indicate the exact nature of the dependency by translating it into class associations between classes of two communicating patterns.

**Pattern language**

A collection of related patterns that work together to develop a solution to an overall, usually large, problem.

**Pattern oriented frameworks**
Design frameworks defined by an architecture of communicating constructional design patterns (in a pattern-level view) which are further extended to classes and objects (in a class level view).

**Pattern-level view**

A schematic diagram that expresses the system design in terms of frameworks, design patterns, and their dependencies.

**Pattern-level with interface view**

A schematic diagram that refines the pattern-level diagram by expressing the pattern interfaces and translating pattern dependencies into interface associations.

**Pattern-oriented design**

A composition of constructional design patterns that interface to form the overall system design.

**POAD (Pattern-Oriented Analysis and Design)**

A new development methodology that creates object-oriented design by gluing together constructional design patterns using their solution structure.

**Responsibility**

An action that an entity takes as part of a collaboration. A responsibility may be carried out in collaboration with other entities.

**Role**

An architectural representation of the objects occupying the corresponding positions in the object system \[\text{Reenskaug 1996}\].

**Role model**

A collaboration of objects that we choose to regard as a unit, separated from the rest of the structure during some period of consideration or analysis.
Scenario

A scenario in an abstract sense is a sequence of component interactions. At the system level, scenarios are activated by the specific input stimulus.

Software library

A controlled collection of software and related documentation designed to aid in software development, use, or maintenance.

Top-down software development

An approach to program development in which progress is made by defining required elements in terms of more basic elements.
Bibliography


[Sommerlad 1996] P. Sommerlad."Command Processor." In J. Vlissides, J. Coplien, & N. Kerth (eds.), Pattern Languages of Program...


Like the book? Buy it!