

THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

Software Engineering

A Methodical Approach

Elvis C. Foster

Apress®

Software Engineering

A Methodical Approach



Elvis C. Foster

Apress®

Software Engineering: A Methodical Approach

Copyright © 2014 by Elvis C. Foster

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0848-9

ISBN-13 (electronic): 978-1-4842-0847-2

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Jim DeWolf

Development Editor: Douglas Pundick

Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade, Steve Weiss

Coordinating Editor: Kevin Walter

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This book is dedicated to my late father, Cladius Foster, who taught me the discipline of being a responsible person. The book is also dedicated to my students — past, present, and future. You are the object of my inspiration and motivation; you are the reason for this book.

Contents at a Glance

About the Author

Acknowledgments

Preface

■ **Part A: Fundamentals**

■ **Chapter 1: Introduction to Software Engineering**

■ **Chapter 2: The Role of the Software Engineer**

■ **Part B: Software Investigation and Analysis**

■ **Chapter 3: Project Selection and the Initial System Requirements**

■ **Chapter 4: The Requirements Specification**

■ **Chapter 5: Information Gathering**

■ **Chapter 6: Communicating Via Diagrams**

■ **Chapter 7: Decision Models for System Logic**

■ **Chapter 8: Project Management Aids**

■ **Part C: Software Design**

■ **Chapter 9: Overview of Software Design**

■ **Chapter 10: Database Design**

■ **Chapter 11: User Interface Design**

■ **Chapter 12: Operations Design**

■ **Chapter 13: Other Design Considerations**

- **Part D: Software Development**
 - **Chapter 14: Software Development Issues**
 - **Chapter 15: Human Resource Management**
 - **Chapter 16: Software Economics**
- **Part E: Software Implementation and Management**
 - **Chapter 17: Software Implementation Issues**
 - **Chapter 18: Software Management**
 - **Chapter 19: Organizing for Effective Management**
- **Part F: Final Preparations**
 - **Chapter 20: Sample Exercises and Examination Questions**
- **Part G: Appendices**
 - **Appendix 1: Introduction to Object Oriented Methodologies**
 - **Appendix 2: Basic Concepts of Object-Oriented Methodologies**
 - **Appendix 3: Object-Oriented Information Engineering**
 - **Appendix 4: Basic Guidelines for Object-Oriented Methodologies**
 - **Appendix 5: Categorizing Objects**
 - **Appendix 6: Specifying Object Behavior**
 - **Appendix 7: Tools for Object-Oriented Methodologies**
 - **Appendix 8: Project Proposal for a Generic Inventory Management System**
 - **Appendix 9: Requirements Specification for a Generic Inventory Management System**
 - **Appendix 10: Design Specification for a Generic Inventory Management System**

Index

Contents

About the Author

Acknowledgments

Preface

■ Part A: Fundamentals

■ Chapter 1: Introduction to Software Engineering

1.1 Definitions and Concepts

 1.1.1 System

 1.1.2 Software

1.2 The Organization as a System

 1.2.1 Discussion

1.3 Information Levels in the Organization

 1.3.1 Top Management

 1.3.2 Middle Management

 1.3.3 Junior Management and Operational Staff

 1.3.4 Importance of Information Levels in Software Engineering

 1.3.5 Alternate Organizational Settings

1.4 Software Life Cycle

 1.4.1 Waterfall Model

 1.4.2 Phased Prototype Model

 1.4.3 Iterative Development Model

 1.4.4 Rapid Prototype Model

 1.4.5 Formal Transformation Model

 1.4.6 Components-Based Model

 1.4.7 Agile Development Model

1.5 Categories of Software

1.6 Alternate Software Acquisition Approaches

1.6.1 Discussion

1.7 Software Engineering Paradigms

1.8 Desirable Features of Computer Software

1.9 Summary and Concluding Remarks

1.10 Review Questions

1.11 References and/or Recommended Readings

■ Chapter 2: The Role of the Software Engineer

2.1 Historical Role

2.2 Modern Role of the Software Engineer

2.3 Job Description of the Software Engineer

2.3.1 Core Functions of the Software Engineer

2.3.2 Desirable Qualities of the Software Engineer

2.4 Tools used by the Software Engineer

2.4.1 Coding Systems

2.4.2 Forms Design

2.4.3 Data Analysis Charts

2.4.4 Technical Documents and Modeling Techniques

2.4.5 Software Planning and Development Tools

2.5 Management Issues with Which the Software Engineer must be Familiar

2.6 Summary and Concluding Remarks

2.7 Review Questions

2.8 References and/or Recommended Readings

■ Part B: Software Investigation and Analysis

■ Chapter 3: Project Selection and the Initial System Requirements

- 3.1 Project Selection
- 3.2 Problem Definition
 - 3.2.1 Constraints of a System
 - 3.2.2 Aid in Identifying System Problems
 - 3.2.3 Identifying the System Void
- 3.3 The Proposed Solution
- 3.4 Scope and Objectives of the System
- 3.5 System Justification
- 3.6 Feasibility Analysis Report
 - 3.6.1 Technical Feasibility
 - 3.6.2 Economic Feasibility
 - 3.6.3 Operational Feasibility
 - 3.6.4 Evaluation of System Alternatives
 - 3.6.5 Evaluation of System Alternatives (continued)
- 3.7 Alternate Approach to the Feasibility Analysis
- 3.8 Summary of System Inputs and Outputs
- 3.9 Initial Project Schedule
- 3.10 Project Team
- 3.11 Summary and Concluding Remarks
- 3.12 Review Questions
- 3.13 References and/or Recommended Readings

■ Chapter 4: The Requirements Specification

- 4.1 Introduction
- 4.2 Contents of the Requirements Specification
- 4.3 Documenting the Requirements
- 4.4 Requirements Validation
- 4.5 How to Proceed

4.6 Presentation of the Requirements Specification

4.7 Summary and Concluding Remarks

4.8 Review Questions

4.9 References and/or Recommended Readings

■ **Chapter 5: Information Gathering**

5.1 Rationale for Information Gathering

5.2 Interviewing

5.3 Questionnaires and Surveys

5.4 Sampling and Experimenting

 5.4.1 Probability Sampling Techniques

 5.4.2 Non-probability Sampling Techniques

 5.4.3 Sample Calculations

5.5 Observation and Document Review

5.6 Prototyping

5.7 Brainstorming and Mathematical Proof

5.8 Object Identification

 5.8.1 The Descriptive Narrative Approach

 5.8.2 The Rule-of-Thumb Approach

5.9 Summary and Concluding Remarks

5.10 Review Questions

5.11 References and/or Recommended Readings

■ **Chapter 6: Communicating Via Diagrams**

6.1 Introduction

6.2 Traditional System Flowcharts

 6.2.1 Information-Oriented Flowchart

 6.2.2 Process-Oriented Flowchart

 6.2.3 Hierarchy- Input–Process–Output Chart

6.3 Procedure Analysis Chart

6.4 Innovation: Topology Charts

 6.4.1 Information Topology Chart

 6.4.2 User Interface Topology Chart

6.5 Data Flow Diagrams

6.6 Object Flow Diagram

6.7 Other Contemporary Diagramming Techniques

 6.7.1 State Transition Diagram

 6.7.2 Finite State Machine

6.8 Program Flowchart

6.9 Summary and Concluding Remarks

6.10 Review Questions

6.11 References and/or Recommended Readings

■ Chapter 7: Decision Models for System Logic

7.1 Structured Language

7.2 Decision Tables

 7.2.1 Constructing the Decision Table

 7.2.2 Analyzing and Refining the Decision Table

 7.2.3 Extended Decision Table

7.3 Decision Trees

7.4 Which Technique to Use

7.5 Decision Techniques versus Flowcharts

7.6 System Rules

 7.6.1 Rule Definition

 7.6.2 Declarative versus Procedural Statements

 7.6.3 Types of Rules

7.7 Summary and Concluding Remarks

7.8 Review Questions

7.9 References and/or Recommended Readings

■ **Chapter 8: Project Management Aids**

8.1 PERT and CPM

- 8.1.1 Step 1: Tabulate the Project Activities
- 8.1.2 Step 2: Draw the PERT Diagram
- 8.1.3 Step 3: Determine ES, EF, LS and LF for each Activity
- 8.1.4 Step 4: Determine the Critical Path
- 8.1.5 Step 5: Conduct a Sensitivity Analysis

8.2 The Gantt Chart

8.3 Project Management Software

8.4 Summary and Concluding Remarks

8.5 Review Questions

8.6 References and/or Recommended Readings

■ **Part C: Software Design**

■ **Chapter 9: Overview of Software Design**

9.1 The Software Design Process

9.2 Design Strategies

- 9.2.1 Function-Oriented Design
- 9.2.2 Object-Oriented Design
- 9.2.3 The Unified Modeling Language
- 9.2.4 Advantages of Object Oriented Design
- 9.2.5 Using Both FO and OO Strategies

9.3 Architectural Design

- 9.3.1. Approaches to Resource Sharing
- 9.3.2 System Controls
- 9.3.3 System Components

9.4 Interface Design

9.5 Software Design and Development Standards

- 9.5.1 Advantages of Software Standards

9.5.2 Issues That Software Standards Should Address

9.6 The Design Specification

9.6.1 Contents of the Design Specification

9.6.2 How to Proceed

9.7 Summary and Concluding Remarks

9.8 Review Questions

9.9 References and/or Recommended Readings

■ Chapter 10: Database Design

10.1 Introduction

10.2 Approaches to Database Design

10.2.1 Conventional Files

10.2.2 Database Approach

10.2.3 Identifying and Defining Entities or Object Types

10.2.4 Identifying Relationships

10.2.5 Developing the ERD or ORD

10.2.6 Preparing the Database Specification

10.3 Overview of File Organization

10.3.1 Sequential File Organization

10.3.2 Relative or Direct File Organization

10.3.3 Indexed Sequential File Organization

10.3.4 Multi-Access File Organization

10.4 Summary and Concluding Remarks

10.5 Review Questions

10.6 References and/or Recommended Readings

■ Chapter 11: User Interface Design

11.1 Introduction

11.1.1 User Needs

11.1.2 Human Factors

11.1.3 Design Considerations

11.1.4 User Interface Preparation

11.2 Types of User Interfaces

11.3 Steps in User Interface Design

11.3.1 Menu or Graphical User Interface

11.3.2 Command-Based User Interface

11.4 Overview of Output Design

11.5 Output Methods versus Content and Technology

11.5.1 Printed Output

11.5.2 Monitor Display

11.5.3 Audio Output

11.5.4 Microfilm/Microfiche

11.5.5 Magnetic and Optical Storage

11.5.6 Choosing the Appropriate Output Method

11.6 Guidelines for Designing Output

11.6.1 Guidelines for Designing Printed Output

11.6.2 Guidelines for Designing Screen Output

11.7 Overview of Input Design

11.8 Guidelines for Designing Input

11.9 Summary and Concluding Remarks

11.10 Review Questions

11.11 References and/or Recommended Reading

■ Chapter 12: Operations Design

12.1 Introduction

12.2 Categorization of Operations

12.3 Essentials of Operations Design

12.4 Informal Methods for Specifying Operation Requirements

12.4.1 Traditional Methods

12.4.2 Warnier Orr Diagram

12.4.3 UML Notations for Object Behavior

12.4.4 Extended Operation Specification

- 12.5 Formal Specifications
- 12.6 Summary and Concluding Remarks
- 12.7 Review Questions
- 12.8 References and/or Recommended Reading

■ **Chapter 13: Other Design Considerations**

- 13.1 The System Catalog
 - 13.1.1 Contents of the System Catalog
 - 13.1.2 Building the System Catalog
 - 13.1.3 Using the System Catalog
- 13.2 Product Documentation
 - 13.2.1 The System Help Facility
 - 13.2.2 The User's Guide and System Guide
- 13.3 User Message Management
 - 13.3.1 Storage and Management of Messages
 - 13.3.2 Message Retrieval
- 13.4 Design for Real-Time Systems
 - 13.4.1 Real-Time System Modeling
 - 13.4.2 Real-Time Programming
- 13.5 Design for Reuse
- 13.6 System Security
 - 13.6.1 Access to the System
 - 13.6.2 Access to System Resources
 - 13.6.3 Access to System Data
- 13.7 Summary and Concluding Remarks
- 13.8 Review Questions
- 13.9 References and/or Recommended Readings

■ **Part D: Software Development**

■ **Chapter 14: Software Development Issues**

14.1 Introduction

14.2 Standards and Quality Assurance

14.2.1 The Relationship between Quality and Standards

14.2.2 Software Quality Factors

14.2.3 Quality Assurance Evaluation

14.3 Management of Targets and Financial Resources

14.3.1 Managing Budget and Expenditure

14.3.2 Managing Software Cost and Value

14.4 Leadership and Motivation

14.5 Planning of Implementation Strategy

14.6 Summary and Concluding Remarks

14.7 Review Questions

14.8 References and/or Recommended Readings

■ Chapter 15: Human Resource Management

15.1 Management Responsibilities

15.2 Management Styles

15.2.1 Autocratic Management

15.2.2 Egalitarian (Democratic) Management

15.2.3 Laissez Faire Management

15.2.4 Path-Goal Leadership

15.2.5 Transformational Leadership

15.2.6 The Super Leader Approach

15.2.7 Task-Oriented Leadership

15.2.8 Relation-Oriented Leadership

15.2.9 Contingency Leadership

15.3 Developing Job Descriptions

15.4 Hiring

15.5 Maintaining the Desired Environment

15.5.1 Effective Communication

15.5.2 Conflict Resolution

15.5.3 Treating Outstanding Achievements and Errant Actions

15.6 Preserving Accountability

15.6.1 Designing and Assigning Work

15.6.2 Evaluating Performance

15.7 Grooming and Succession Planning

15.8 Summary and Concluding Remarks

15.9 Review Questions

15.10 References and/or Recommended Readings

■ Chapter 16: Software Economics

16.1 Software Cost versus Software Price

16.1.1 Software Cost

16.1.2 Software Price

16.2 Software Value

16.3 Evaluating Software Productivity

16.3.1 Size-related Metrics

16.3.2 Function-related Metrics

16.3.3 Assessment Based on Value Added

16.4 Estimation Techniques for Engineering Cost

16.4.1 Algorithmic Cost Models

16.4.2 The COCOMO Model

16.4.3 The COCOMO II Model

16.5 Summary and Concluding Remarks

16.6 Review Questions

16.7 References and/or Recommended Reading

■ Part E: Software Implementation and Management

■ Chapter 17: Software Implementation Issues

17.1 Introduction

17.2 Operating Environment

- 17.2.1 Central System
- 17.2.2 Distributed System
- 17.2.3 Other Environmental Issues

17.3 Installation of the System

17.4 Code Conversion

17.5 Change Over

- 17.5.1 Direct Change Over
- 17.5.2 Parallel Conversion
- 17.5.3 Phased Conversion
- 17.5.4 Distributed Conversion

17.6 Training

17.7 Marketing of the Software

17.8 Summary and Concluding Remarks

17.9 Review Questions

17.10 Recommended Readings

■ Chapter 18: Software Management

18.1 Introduction

18.2 Software Maintenance

- 18.2.1 Software Modifications
- 18.2.2 Software Upgrades and Patches
- 18.2.3 Maintenance Cost

18.3 Legacy Systems

18.4 Software Integration

18.5 Software Re-engineering

18.6 Summary and Concluding Remarks

18.7 Review Questions

18.8 References and/or Recommended Readings

■ Chapter 19: Organizing for Effective Management

- [19.1 Introduction](#)
- [19.2 Functional Organization](#)
- [19.3 Parallel Organization](#)
- [19.4 Hybrid \(Matrix\) Organization](#)
- [19.5 Organization of Software Engineering Firms](#)
- [19.6 Summary and Concluding Remarks](#)
- [19.7 Review Questions](#)
- [19.8 References and/or Recommended Readings](#)

■ Part F: Final Preparations

■ Chapter 20: Sample Exercises and Examination Questions

- [20.1 Introduction](#)
- [20.2 Assignment 1A](#)
- [20.3 Assignment 2](#)
- [20.4 Assignment 3A](#)
- [20.5 Assignment 4A](#)
- [20.6 Assignment 5A](#)
- [20.7 Assignment 6A](#)
- [20.8 Assignment 7A](#)
- [20.9 Assignment 8A](#)
- [20.10 Sample Interim Examination 1A](#)
- [20.11 Sample Interim Examination 2B](#)
- [20.12 Sample Final Examination A](#)
- [20.13 Sample Final Examination B](#)

■ Part G: Appendices

■ Appendix 1: Introduction to Object Oriented Methodologies

A1.1 Software Revolution and Rationale for Object-Oriented Techniques

A1.2 Information Engineering and the Object-Oriented Approach

A1.3 Integrating Hi-tech Technologies

A1.4 Characteristics of Object-Oriented Methodologies

A1.5 Benefits of Object-Oriented Methodologies

A1.6 Summary and Concluding Remarks

A1.7 Recommended Readings

■ Appendix 2: Basic Concepts of Object-Oriented Methodologies

A2.1 Objects and Object Types

A2.2 Operations

A2.3 Methods

A2.4 Encapsulation and Classes

 A2.4.1 Encapsulation

 A2.4.2 Class

A2.5 Inheritance and Amalgamation

 A2.5.1 Inheritance

 A2.5.2 Amalgamation

A2.6 Requests

A2.7 Polymorphism and Reusability

A2.8 Interfaces

A2.9 Late Binding

A2.10 Multithreading

A2.11 Perception versus Reality

A2.12 Overview of the Object-Oriented Software Engineering Process

A2.13 Summary and Concluding Remarks

A2.14 Recommended Readings

■ **Appendix 3: Object-Oriented Information Engineering**

A3.1 Introduction

A3.2 Engineering the Infrastructure

A3.3 Diagramming Techniques

A3.4 Enterprise Planning

A3.5 Business Area Analysis

A3.6 System Design

A3.7 System Construction

A3.8 Summary and Concluding Remarks

A3.9 References and/or Recommended Reading

■ **Appendix 4: Basic Guidelines for Object-Oriented Methodologies**

A4.1 Object Identification

 A4.1.1 Using Things to be Modeled

 A4.1.2 Using the Definitions of Objects, Categories and Interfaces

 A4.1.3 Using Decomposition

 A4.1.4 Using Generalizations and Subclasses

 A4.1.5 Using OO Domain Analysis or Application Framework

 A4.1.6 Reusing Hierarchies, Individual Objects and Classes

 A4.1.7 Using Personal Experience

 A4.1.8 Using the Descriptive Narrative Approach

 A4.1.9 Using the Class-Responsibility-Collaboration Method

 A4.1.10 Using the Rule-of-Thumb Method

A4.2 End User Involvement

A4.3 OO Diagramming

A4.4 Enterprise-wide Design

A4.5 Emphasis on OO-CASE Tools versus OO-Programming Language

A4.6 OO Modeling

A4.7 Summary and Concluding Remarks

A4.8 References and/or Recommended Reading

■ Appendix 5: Categorizing Objects

A5.1 Identifying Object Relationships

A5.2 Fern Diagram

A5.3 Information Topology Chart

A5.4 Object Relationship Diagrams

A5.5 Representing Details about Object Types

 A5.5.1 Object Structure Diagram

 A5.5.2 CRC Card

A5.6 Avoiding Multiple Inheritance Relationships

 A5.6.1 Delegation Using Aggregation

 A5.6.2 Delegation and Inheritance

 A5.6.3 Nested generalization

A5.7 Top-Down versus Bottom-Up

 A5.7.1 Top-Down Approach

 A5.7.2 Bottom-Up Approach

A5.8 Summary and Concluding Remarks

A5.9 References and/or Recommended Reading

■ Appendix 6: Specifying Object Behavior

A6.1 Use-cases

 A6.1.1 Symbols Used in Use-case

 A6.1.2 Types of Use-cases

 A6.1.3 Information Conveyed by a Use-case

 A6.1.4 Bundling Use-cases and Putting Them to Use

- A6.2 States and State Transition
- A6.3 Finite State Machines
- A6.4 Event Diagrams
- A6.5 Triggers
- A6.6 Activity Diagrams
- A6.7 Sequence Diagrams and Collaboration Diagrams
- A6.8 Object Flow Diagrams
- A6.9 Summary and Concluding Remarks
- A6.10 References and/or Recommended Reading

■ **Appendix 7: Tools for Object-Oriented Methodologies**

- A7.1 Introduction
- A7.2 Categories of CASE Tools
- A7.3 Universal Database Management Systems
- A7.4 Benefits of OO-CASE Tools and UDBMS Suites
- A7.5 Object Oriented Programming Languages
- A7.6 Modeling and Code Generation
 - A7.6.1 Instant CASE
 - A7.6.2 Repository
- A7.7 Standards for OOM
 - A7.7.1 CORBA
 - A7.7.2 COM
 - A7.7.3 The .NET Boom
- A7.8 Summary and Concluding Remarks
- A7.9 References and/or Recommended Readings

■ **Appendix 8: Project Proposal for a Generic Inventory Management System**

- A8.1 Problem Definition

- A8.2 Proposed Solution
- A8.3 Scope of the System
- A8.4 System Objectives
- A8.5 Expected Benefits
- A8.6 Overview of Storage Requirements
- A8.7 Anticipated Outputs
- A8.8 Feasibility Analysis Report
 - A8.8.1 Feasibility of Alternative A
 - A8.8.2 Feasibility of Alternative B
 - A8.8.3 Feasibility of Alternative C
 - A8.8.4 Feasibility of Alternative D
 - A8.8.5 Evaluation of Alternatives
- A8.9 Initial Project Schedule

■ **Appendix 9: Requirements Specification for a Generic Inventory Management System**

- A9.1 System Overview
 - A9.1.1 Problem Definition
 - A9.1.2 Proposed Solution
 - A9.1.3 System Architecture
- A9.2 Storage Requirements
 - A9.2.1 Acquisitions Management Subsystem
 - A9.2.2 Financial Management Subsystem
- A9.3 Operational Requirements
- A9.4 Business Rules
 - A9.4.1 Overview
 - A9.4.2 Derivation and Procedural Rules
- A9.5 Summary and Concluding Remarks

■ **Appendix 10: Design Specification for a Generic Inventory Management System**

A10.1 System Overview

A10.1.1 Problem Definition

A10.1.2 Proposed Solution

A10.1.3 System Architecture

A10.2 Database Specification

A10.2.1 Introduction

A10.2.2 Acquisitions Management Subsystem

A10.2.3 Financial Management Subsystem

A10.2.4 Systems Control Subsystem

A10.3 Operations Specification

A10.3.1 Generic Pseudo-codes

A10.3.2 Acquisitions Management Subsystem

A10.3.3 Financial Management Subsystem

A10.3.4 System Controls Subsystem

A10.4 User Interface Specification

A10.5 Message and Help Specifications

A10.5.1 Message Specification

A10.5.2 Help Specification

A10.6 Summary and Concluding Remarks

Index

About the Author



Elvis C. Foster is Associate Professor of Computer Science at Keene State College, New Hampshire. He holds a Bachelor of Science (BS.) in Computer Science and Electronics, as well as a Doctor of Philosophy (PhD) in Computer Science (specializing in strategic information systems and database systems) from University of the West Indies, Mona Jamaica. Dr. Foster has over 25 years of combined experience as a software engineer, information technology executive and consultant, and computer science educator. He has had the favorable experience of being involved with the design and/or development of software systems for various medium-sized and large organizations, including the central bank of his own country. He has lectured at the tertiary level in three different countries, including the United States, and has produced many outstanding computer science and information technology professionals. Many of them have excelled at graduate school, and are doing well in leading software engineering enterprises around the world. This text draws from that experience.

Acknowledgments

My profound gratitude is owed to my wife, Jacqueline, and children Chris-Ann and Rhoden, for putting up with me during the periods of preparation of this text. Also, I must recognize several of my past and current students (from four different institutions and several countries) who at various stages have encouraged me to publish my notes, and have helped to make it happen. In this regard, I would like to make special mention of Dionne Jackson, Kerron Hislop, Brigid Winter, Sheldon Kennedy, Ruth Del Rosario, Brian Yap, and Rossyl Lashley.

I offer a big thank you to Dr. Han Reichgelt at Southern Polytechnic State University, who in many ways has been my professional mentor. As on previous occasions, I have relied on him for critical evaluations and advice. Speaking of critical evaluations, the contribution of my relatively new friend and colleague, Dr. Jared Bruckner at Southern Adventist University was also significant. In the same way that Dr. Han Reichgelt has mentored me during the advanced stages of my career as a computer science professional, Dr. Ezra Mugissa deserves mention for the early formative years of my professional journey. In many respects, my methodical approach to software engineering is owed to him.

The editorial and production teams at Xlibris Corporation deserve mention for their work in facilitating initial publication of the volume. An equally significant level of gratitude is extended to the editorial team at Apress Publishing for recognizing the work's value and for investing the time and effort in the project. Thanks to everyone involved.

Finally, I should also make mention of reviewers Jared Bruckner, Marlon Moncrieffe, Jacob Mangal, Abrams O'Buyonge, and Han Reichgelt, each a practicing software engineer, information technology consult, or computer science professor who has taken time to review the manuscript and provide useful feedback. Thanks, gentlemen.

—Elvis C. Foster, PhD
Keene State College
Keene, New Hampshire, USA

Preface

This book has been compiled with three target groups in mind: The book is best suited for undergraduate students who are pursuing a course in software engineering. Graduate students who are pursuing an introductory course in the subject will also find it useful. Finally, practicing software engineers who need a quick reference on various software engineering methodologies, may also find it useful.

The motivation that drove this work was a desire to provide a concise but comprehensive guide to the discipline of software engineering. Having worked in the information technology (IT) industry for several years, before making a career switch to academia, and having myself witnessed the struggles of many new entrants to the discipline of software engineering, I have upheld the view for some time that a concise but comprehensive reference in software engineering would be useful to students as well as practitioners in the industry.

These notes have been compiled and tested over several years with outstanding results. They draw on personal experiences gained in industry over the years, as well as the suggestions of various professionals and students. The chapters are organized in a manner that reflects my own approach in lecturing the course, but each chapter may be read on its own merit.

The text has been prepared specifically to meet three objectives: comprehensive coverage, brevity, and relevance.

Comprehensive coverage and brevity often operate as competing goals. In order to achieve both, I have adopted a methodical and pragmatic approach that gets straight to the critical issues for each topic, and avoids unnecessary fluff, while using the question of relevance as the balancing force. Additionally, readers should find the following features quite convenient and reader friendly:

- Short paragraphs that express the salient aspects of the subject matter being discussed
- Bullet points or numbers to itemize important things to be remembered
- Diagrams and illustrations to enhance the reader's understanding

- Overview and summary of each chapter
- Introduction of a number of original software engineering methodologies
- Discussion of solutions to generic software engineering problems in a step-by-step manner
- A chapter with sample examination questions (for the student) and case studies (for the student as well as the inexperienced software engineer)
- Also, my introduction of original methodologies for treating certain software engineering problems should make a useful contribution to already existing literature on the subject. These new methodologies include (but are not confined to) the following: information topology chart (ITC) and user interface topology chart (UITC) in [chapters 6 and 11](#), object/entity specification grid (O/ESG) in [chapter 10](#), extended operation specification (EOS) in [chapter 12](#), and a generic object-naming convention for any software engineering project in [chapter 9](#) and [appendix 10](#).

Organization of the Text

The text is organized in nineteen chapters plus a twentieth chapter consisting of sample examination questions and case studies. The chapters are placed into six divisions. There is also a seventh division that includes appendices in several topics of interest. The chapters and related divisions follow:

[Part A: Introductory Topics](#)

[Chapter 1](#), Introduction to Software Engineering: This chapter introduces you to the discipline of software engineering. It defines software engineering, and discusses its importance in the context of a business organization, and as a dominant field of computer science.

[Chapter 2](#), The Role of the Software Engineer: This chapter discusses the role, responsibilities, and tools used by the software engineer.

Part B: Software Investigation and Analysis

[Chapter 3](#), Project Selection and Initial System Requirement: Here, the preliminary activities and the first deliverable of a software engineering project are discussed.

[Chapter 4](#), The Requirements Specification: This chapter discusses the second deliverable of a software engineering project.

[Chapter 5](#), Information Gathering: Various techniques for obtaining useful information about the requirements of a software system are discussed in this chapter.

[Chapter 6](#), Communication via Diagrams: This chapter discusses various diagramming techniques that are available to the software engineer.

[Chapter 7](#), Decision Models for System Logic: This continues the discussion of diagramming techniques by focusing on methodologies for representing system logic.

[Chapter 8](#), Project Management Aids: Here, the focus is on PERT, CPM, Gant Chart, and project management software systems.

Part C: Software Design

[Chapter 9](#), Overview of Software Design: This chapter briefly clarifies the various aspects of software design, and shows how they converge into a significant deliverable of a software engineering project —the design specification.

[Chapter 10](#), Database Design: This chapter discusses database design as an important component of software design. It summarizes the salient aspects of database design, but emphasizes that further study of this topic is absolutely necessary.

[Chapter 11](#), User Interface Design: Here, the user interface is presented as an essential window through which end-users access and interact with the software system. The fundamentals of user interface design are discussed.

[Chapter 12](#), Operations Design: Various methodologies for operations design are discussed in this chapter. The focus is primarily on informal methods.

[Chapter 13](#), Other Design Considerations: This chapter addresses other design issues not covered in the previous four.

Part D: Software Development

[Chapter 14](#), Software Development Issues: This chapter introduces software development as an exciting experience, if and when it follows careful software planning and design. It identifies various issues to be addressed during software development.

[Chapter 15](#), Human Resource Management: Here, effective human resource management (HRM) is discussed as an essential aspect of good software engineering.

[Chapter 16](#): Software Economics: In this chapter, focus is placed on techniques employed in the determination of the cost, price, and value of software systems.

Part E: Software Implementation and Management

[Chapter 17](#), Software Implementation Issues: This chapter discusses various approaches to software implementation.

[Chapter 18](#), Software Management: In this chapter, various aspects related to the management of software systems are discussed.

[Chapter 19](#), Organizing for Effective Management: This chapter provides various options for organizing a software engineering enterprise.

Part F: Final Preparations

[Chapter 20](#), Sample Exercises and Examination Questions

Part G: Appendices

The appendices provide additional discussions and illustrations for your usage. Appendices 1 – 7 provide discussions and additional insights on the object-oriented approach to software engineering. Appendices 8 – 10 provide excerpts from the specification and design of a generic inventory management system. These documents illustrate how some of the principles covered in the course are applied in a real project.

The topics of the appendices are as follows:

- [Appendix 1](#): Introduction to Object-Oriented Methodologies
- [Appendix 2](#): Basic Concepts of Object-Oriented Methodologies
- [Appendix 3](#): Object-Oriented Information Engineering
- [Appendix 4](#): Basic Guidelines of Object-Oriented Methodologies
- [Appendix 5](#): Categorizing Objects
- [Appendix 6](#): Specifying Object Behavior
- [Appendix 7](#): Tools for Object-Oriented Methodologies
- [Appendix 8](#): Excerpts from the ISR of the Inventory Management System
- [Appendix 9](#): Excerpts from the RS of the Inventory Management System
- [Appendix 10](#): Excerpts from the DS of the Inventory Management System

Text Usage

The text could be used as a one-semester or two-semester course in software engineering, augmented with an appropriate CASE or RAD tool. Below are two suggested schedules for using the text. One assumes a one-semester course; the other assumes a two-semester course. The schedule for a one-semester course is a very aggressive one that assumes adequate preparation on the part of the participants. The schedule for a two-semester course gives the participants more time to absorb the material, and gain mastery in the various methodologies discussed by engaging in a meaningful project. This obviously, is the preferred scenario.

One-Semester Schedule:	
Week	Topic
01	Chapter 01
02	Chapter 02
03	Chapter 03
04	Chapters 04 & 05
05	Chapter 06
06	Chapter 06 & 07
07	Chapter 08
08	Chapter 09
09	Chapter 10
10	Chapters 11 & 12
11	Chapters 12 & 13
12	Chapters 14 & 15
13	Chapter 16
14	Chapters 17 & 18
15	Chapter 19
16	Review

Two-Semester Schedule:	
Week	Topic
01	Chapter 01
02	Chapter 02
03	Chapter 03
04	Chapters 04 & 05
05	Chapter 06
06	Chapter 06 & 07
07	Chapter 08
08	Chapter 09
09	Chapter 10
10	Chapter 11
11	Chapter 12
12	Chapter 13
13	Chapters 14 & 15
14	Chapter 16
15	Chapter 17
16	Chapters 18 & 19
17	Review
18	Review
19-32	Course Project

Approach

Throughout the text, I have adopted a practical, methodical approach to software engineering, avoiding an overkill of theoretical calculations where possible (these can be obtained elsewhere). The primary objective is to help the reader to gain a good grasp of the activities in the software development life cycle (SDLC). At the end of the course, the participant should feel confident about taking on a new software engineering project.

Feedback and Support

It is hoped that you will have as much fun using this book as I had preparing it. Additional support information can be obtained from the Web site <http://www.elcfos.com> or <http://www.elcfos.net>. Also, your comments will be appreciated.

PART A



Fundamentals

This preliminary division of the course is designed to cover some fundamentals. The objectives are as follows:

- To define and provide a rationale for software engineering as a discipline
- To provide you with a wide perspective of computer software and its varied usefulness and applications
- To discuss different approaches to software acquisition
- To define the job of the software engineer in the organization
- To discuss various tools used by the software engineer

The division consists of two chapters:

- [Chapter 1](#) — Introduction to Software Engineering
- [Chapter 2](#) — The Role of the Software Engineer

[REDACTED]

CHAPTER 1



Introduction to Software Engineering

Welcome and congratulations on your entry to this course in software engineering. The fact that you are in this course means that you have covered several fundamental topics in programming, data structures, and perhaps user interface. You have been writing computer programs to solve basic and intermediate-level problems. Now you want to take your learning experience to another level: you want to learn how to design, develop and manage complex software systems (small, medium sized and large), which may consist of tens or hundreds of programs all seamlessly integrated into a coherent whole. You will learn all of these and more in this course, but first, we must start at the beginning. This chapter introduces you to the discipline of software engineering. Topics covered include the following:

- Definitions and Concepts
- The Organization as a System
- Information Levels in the Organization
- Software Life Cycle
- Categories of Software
- Alternate Software Acquisition Approaches
- Software Engineering Paradigms
- Desirable Features of Computer Software
- Summary and Concluding Remarks

1.1 Definitions and Concepts

Computer software affects almost all aspects of human life. A study of the process of software construction and management is therefore integral to any degree in computer science, computer information systems, multimedia technology, or any other related field. Software systems are not created, and do not exist in a vacuum; rather they are typically created by individuals and/or organizations, for use by individuals and/or organizations. We will therefore start by defining a system, defining software, identifying the relationship between the two, and then showing how they both relate to the organization.

1.1.1 System

A system is a set of interacting, interrelated, interdependent components that function as a whole to achieve specific objectives. An effective system must be synergistic. The system usually operates in an environment external to itself. A system may also be defined as the combination of personnel, materials, facilities and equipment working together to convert input into meaningful and needed outputs.

Following are some fundamental principles about systems:

- The components of a system are interrelated and interdependent.
- The system is usually viewed as a whole.
- Every system has specific goals.
- There must be some type of inputs and outputs.
- Processes prescribe the transformation of inputs to outputs.
- Systems exhibit entropy, i.e., tendency to become disorganized.
- Systems must be regulated (planning, feedback and control).
- Every system has subsystems.
- Systems exhibit a tendency to a final state.

As a personal exercise, you are encouraged to identify examples of systems in areas with which you are familiar. A good place to start is the human body.

1.1.2 Software

Software is the combination of program(s), database(s) and documentation in a systemic suite, and with the sole purpose of solving specific system problems and meeting predetermined objectives. Software adds value to the hardware components of a computer system. In fact, without software, a computer is reduced to nothing more than an electronic box of no specific use to most human beings. Also, it should not surprise you that computer software is a special kind of system. This course will teach you how to design, construct and manage such systems.

Software Engineering

Software engineering is the process by which software systems are investigated, planned, modeled, developed, implemented and managed. It also includes the re-engineering of existing systems with a view to improving their role, function, and performance. The ultimate objective is the provision or improvement of desirable conveniences and the enhancement of productivity within the related problem domain.

System transformation may take various paths, some of which may be:

- Improving the internal workings of the system
- Modifying inputs and outputs
- Modifying the goals and objectives of the system
- Redesigning the system
- Designing and developing a new system based on existing problems

Steps in the Analysis Process

Before embarking on any major work in software engineering, a process of research and analysis takes place. This process may be summarized in the following steps:

1. Define the problem.

2. Understand the problem (system) — the interrelationships and interdependencies; have a picture of the variables at work within the system; define the extent of the system (problem).
3. Identify alternate solutions.
4. Examine alternate solutions.
5. Choose the best alternative.
6. Pursue the chosen alternative.
7. Evaluate the impact of the (new/modified) system.

1.2 The Organization as a System

Traditionally, many software systems were created by organizations for use in organizations. To a large extent, this scenario still holds true. For the moment, let us therefore take a look at the organization, in the context of this approach. An organization is a collection of individuals that employ available facilities, resources and equipment to work in pursuit of a predetermined set of objectives. The organization qualifies as a system since it has all the ingredients in the definition of a system: people, facilities, equipment, materials and methods of work.

Every organization has certain functional areas (called divisions, departments, sections, etc.). Typical areas include finance and planning, human resource management (HRM), marketing, production and operations (or the equivalent), technical/manufacturing (or the equivalent). These are usually further divided into departments, units and sub-units. Traditionally, a data processing department/unit would be enlisted as a sub-unit of finance. However, more enlightened organizations are now enlisting information technology (IT) as a functional area at the senior management level, servicing all other areas.

[Figure 1-1](#) shows what a highly summarized organizational chart for a modern organization might look like. In tiny or small organizations, each unit that appears under the president or chief executive officer (CEO) may be a department. In medium-sized and large organizations, each unit under the president or CEO is typically a division, consisting of several departments and/or sections. It should also be noted that you are unlikely to find IT (or the equivalent) at the senior level in many traditional organizations, as managers still

struggle with appreciating the scope and role of IT in the organization. However, in more progressive and forward-thinking organizations, you will find that IT is correctly and appropriately positioned at the senior level of the management hierarchy.

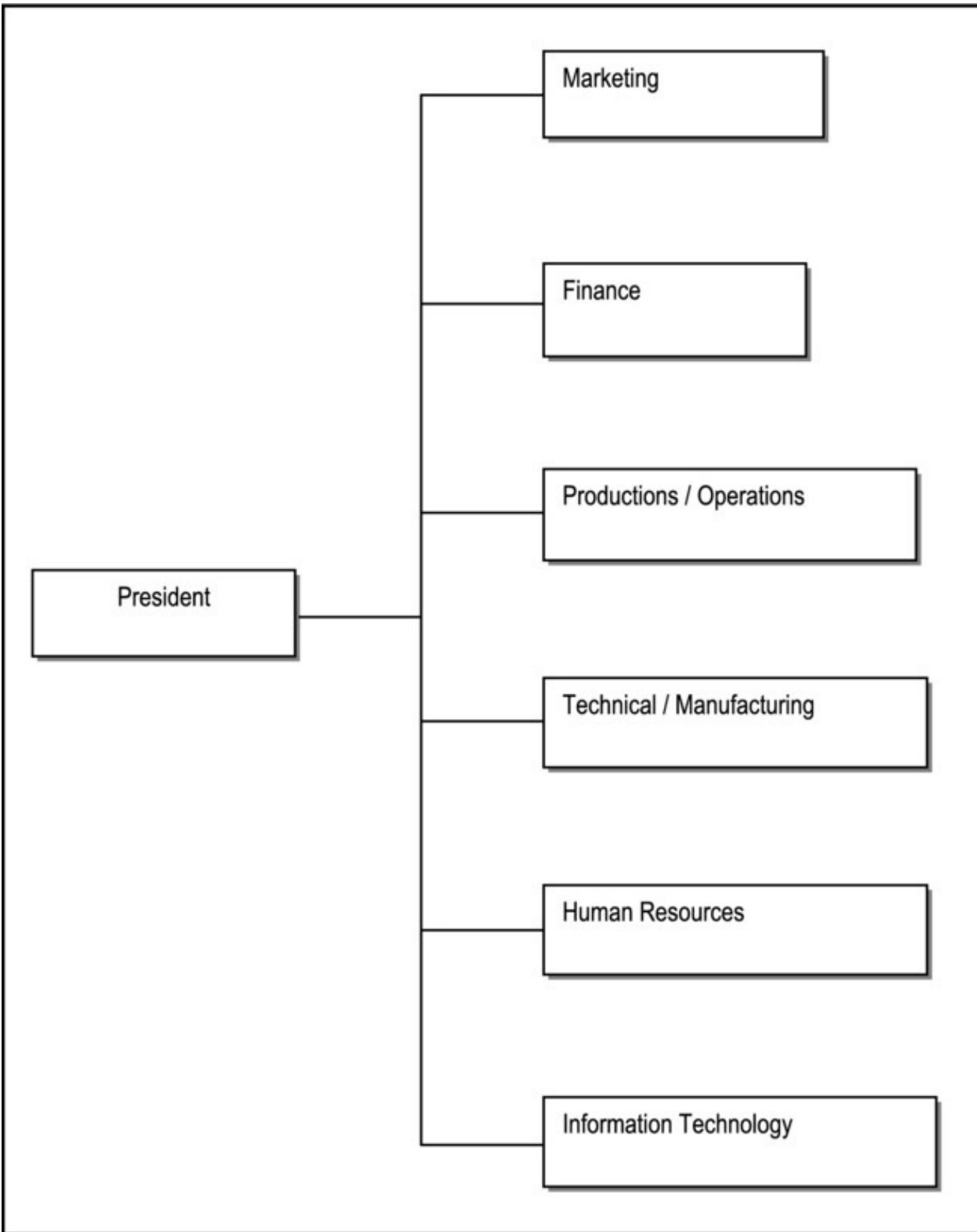


Figure 1-1. Typical Organizational Chart

1.2.1 Discussion

1.2.1 DISCUSSION

Why should IT (or its equivalent) be positioned at the senior level in the organization? If the answer to this question is not immediately obvious to you, it will be by the time you complete this course.

1.3 Information Levels in the Organization

[Figure 1-2](#) shows the information levels in an organization. Information flows vertically as well as horizontally. Vertically, channels are very important. The chart also summarizes the characteristics and activities at each level. Let us examine these a bit closer:

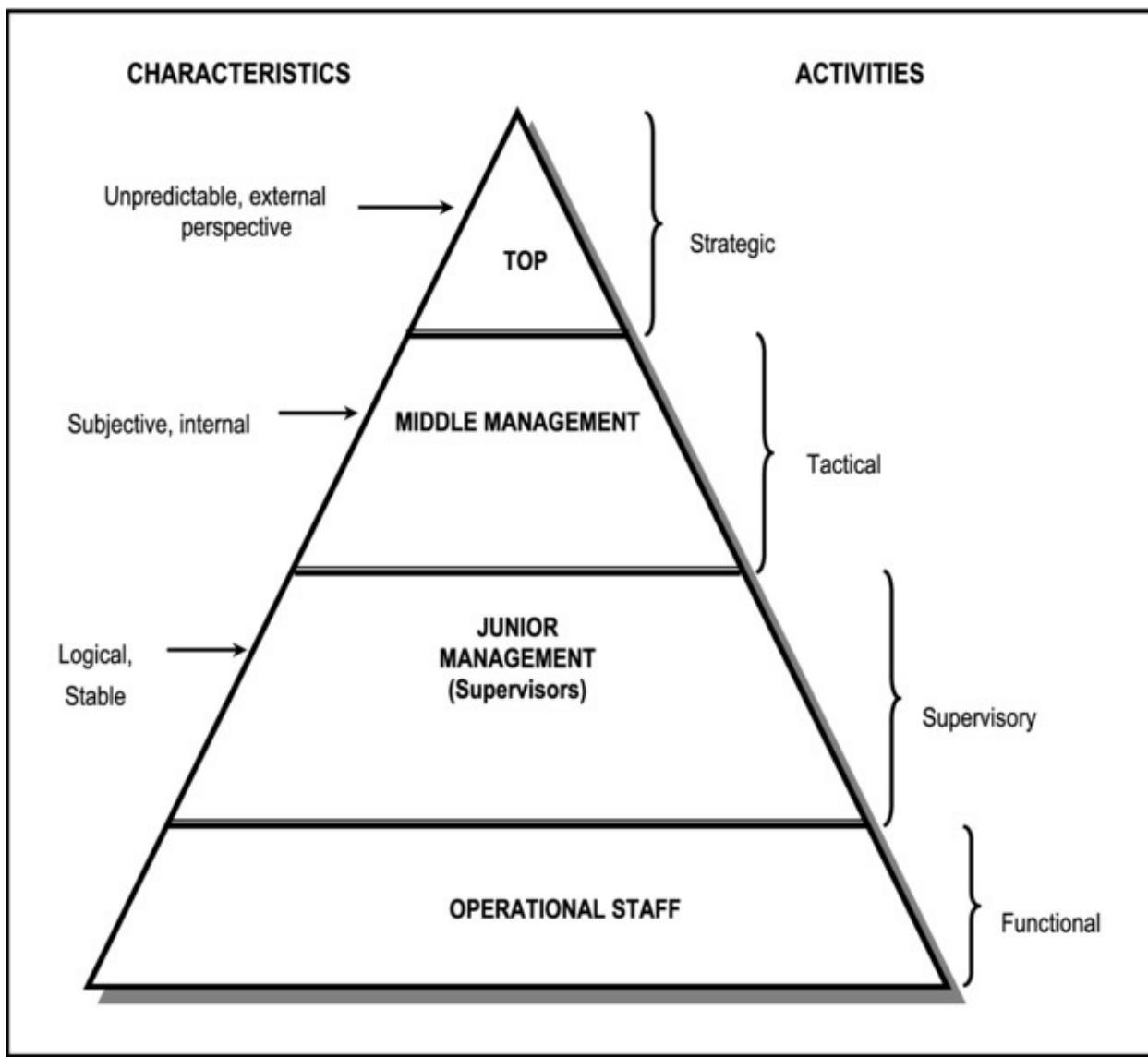


Figure 1-2. Information levels in the organization

1.3.1 Top Management

Activities are strategic and may include:

- Goal setting
- Long term planning
- Expansion or contraction or consolidation
- Merging

- Other strategic issues

For the individuals that operate at this level, there is always an external perspective in the interest of organizational image and interest.

1.3.2 Middle Management

Activities are of a tactical nature and may include:

- Allocation and control of resources
- Medium range planning
- Delegation
- Performance measurement and analysis

For the individuals that operate at this level, subjectivity is rather high — one's own style is brought into play. There is an internal perspective.

1.3.3 Junior Management and Operational Staff

At the junior management level, one is mainly concerned with:

- Job scheduling
- Checking operational results
- Maintaining discipline and order

Operations at the junior management level are very logical and predictable.

The lowest level is the operational staff. The individuals who work at this level carry out the daily routine activities that keep the organization functioning.

1.3.4 Importance of Information Levels in Software Engineering

The information levels are of importance to the software engineer for two reasons:

- Information gathering and analysis must span the entire organization; therefore communication at the appropriate level is important.
- The information needs vary with each level. An effective software system must meet the need at each level.

Discussion

What kind of information would be required at each level? Propose an integrated software system (showing main subsystems) for a university, a variety store, and a hardware store respectively.

1.3.5 Alternate Organizational Settings

Not all software systems are developed within a structured setting as portrayed in the foregoing discussion. In many cases, software systems are developed by software engineering firms, and marketed to the consuming public. These organizations tend to be more flexible in the way they are structured, often favoring more highly skilled employees, and a more flattened structure. This approach is more fully discussed in [chapter 19](#).

Software systems may also be constructed by amorphously structured organizations. This classification includes individuals operating independently, or as collaborative groups. The open source community is an excellent example of this kind of amorphous operation.

Whatever the circumstance, each software system is only relevant if it fulfills a need that people recognize. It must help solve a problem, and it typically has a period of relevance.

1.4 Software Life Cycle

Every software system has a life cycle — a period over which it is investigated/conceived, designed, development and remains applicable or needed. Various life cycle models have been proposed; we shall examine seven:

- Waterfall Model
- Phased Prototype Model
- Iterative Development Model
- Rapid Prototype Model
- Formal Transformation Model
- Component-Based Model
- Agile Development Model

Irrespective of the model used, however, a software system passes through the five phases, as depicted in [figure 1-3](#) (related deliverables also shown). These phases constitute the *software development life cycle* (SDLC).

SDLC Phase	Related Deliverable(s)
Investigation and Analysis	Initial System Requirements; Requirements Specification
Design (Modeling)	Design Specification
Development (Construction)	Actual Software System; Product Documentation
Implementation	
Management	Enhanced Software System; Revised Documentation

Figure 1-3. SDLC Phases and Related Deliverables

1.4.1 Waterfall Model

The waterfall model is the traditional approach to software engineering; it may be summarized in the following points:

- It assumes that total knowledge of the requirements of a system can be obtained before its development.
- Each phase of the system life cycle is signed off with the users before advancing to the next phase.
- The process is irreversible - retracting is not allowed until the system is completed.

The model has the following advantages:

- It ensures a comprehensive, functional, well-integrated system.
- It minimizes the level user complaints.
- It ensures user participation (since the user must signoff on each phase).
- It is likely to result in a well-documented system.

The model is not void of major disadvantages:

- System development is likely to take a long time; users may become impatient.
- The requirements of the system may change before the system is completed.
- One may therefore have a well-designed, well-documented system that is not being used, due to its irrelevance.

1.4.2 Phased Prototype Model

This model (also referred to as the *evolutionary development* model) may be summarized in the following steps:

1. Investigate and fully define the system, identifying the major components.
2. Take a component and model it; then develop and implement it.
3. Obtain user feedback.
4. Revise the model if necessary.
5. If the system is not completed, go back to step 2.

The advantages of phased prototype model are:

- The user gets a feel of the system before its completion.
- Improved user participation over the waterfall model.

- The likelihood of producing an acceptable system is enhanced.

The disadvantages of the phased prototype model include:

- The increased likelihood of a poorly documented system.
- The system may be poorly integrated.
- The system will therefore be more difficult to maintain.

1.4.3 Iterative Development Model

The iterative development model is in some respects a refinement of the phased prototype model. In this approach, the entire life cycle is composed of several iterations. Each iteration is a mini-project in and of itself, consisting of the various lifecycle phases (investigation and analysis, design, construction, implementation, and management). The iterations may be in series or in parallel, but are eventually integrated into a release of the project. The final iteration results in a release of the complete software product.

The advantages of the iterative development model are identical to those of the phased prototype model. However, due to the precautions inherent in the approach, disadvantages (of the phased prototype model) are minimized.

In many respects, the iterative development approach to software construction has been immortalized by the *Rational Unified Process* (RUP). RUP is an iterative development lifecycle framework that has become a very popular in the software engineering industry. RUP was first introduced by the pioneers of Rational Software — Grady Booch, James Rumbaugh and Ivar Jacobson. Since 2003, the company has been acquired by International Business Machines (IBM). IBM currently markets the Rational product line as one of its prime product lines.

1.4.4 Rapid Prototype Model

Rapid prototyping is a commonly used (perhaps overused) term in software engineering. It refers to the rapid development of software via the use of sophisticated software development tools, typically found in CASE (*computer aided software engineering*) tools and DBMS (*database management system*)

suites. Another term that keeps flying around is *rapid application development* (RAD).

The line of distinction of a RAD tool from a CASE tool is not always very clear: in both cases, we are talking about software systems that facilitate development of other software systems by providing among others, features such as:

- Automatic generation of code
- Convenient, user friendly (and typically graphical) user interface
- Executable diagrams

To further blur the distinction, contemporary DBMS suites provide those features also. These tools will be further discussed in [chapter 2](#).

The rapid prototype model may be summarized in the following steps:

- Obtain an idea of the system requirements from user.
- Develop a prototype (possibly under the observation of the user).
- Obtain user feedback.
- Revise if necessary.

One point of clarification: RAD tools, CASE tools, DBMS suites and the like may be employed in any software engineering project, irrespective of the model being followed. Rapid prototyping describes a process, not the tools used.

Rapid prototyping provides us with two significant advantages:

- The system developed is obtained quickly if the first prototype is correct.
- The approach is useful in the design of *expert systems* as well as small end-user applications.

The main disadvantages of rapid prototyping are the following:

- The system may be poorly documented.
- The system may be difficult to maintain.

- System development could take long if the prototypes are wrong.

1.4.5 Formal Transformation Model

The formal transformation model produces software from mathematical system specifications. These transformations are “correctness preserving” and therefore ensure software quality. A number of formal specification languages have been proposed. However, much to the chagrin of its proponents, software development via this method is not as popular as hoped. Formal methods will be further discussed in [chapter 12](#).

The main advantage of the formal transformation model is the production of *provable software* i.e. software generated based on sound mathematical principles. This means that the reliability and quality of the software is high.

The model suffers from three haunting disadvantages:

- The approach (of formal methods) is not always relevant to the problem domain.
- The approach uses abstract specifications with which the software engineer must become familiar.
- Due to the use of quite abstract notations, end user participation is not likely to be high, thus violating an essential requirement for software acceptance.

1.4.6 Components-Based Model

The component-based approach produces software by combining tested and proven components from other software products. As the discipline of software engineering becomes more established and more software standards are established, this approach is expected to be more widely used. The approach is commonly called *component-based software engineering* (CBSE).

The main advantages of CBSE are the following:

- Improvement in the quality and reliability of software
- Software construction can be faster

The main disadvantage of the approach is that like the formal transformation model, it is not always relevant to the problem domain. The reason for this is that software engineering, being a relatively new discipline, has not established enough standards for solving the many and varied software needs that are faced by the world.

1.4.7 Agile Development Model

The agile development model pulls ideas from phased prototyping, iterative development, and rapid prototyping into a model that champions the idea of emphasizing the results of the software engineering effort over the process of getting to the results. The traditional approach of investigation and analysis, design, development, implementation, and management is deemphasized. In contrast, the agile approach emphasizes construction and delivery.

Agile development calls for small, highly talented, highly responsive teams that construct software in small increments, focusing on the essential requirements. The chief architects of the methodology have articulated 12 principles that govern the agile development approach [Beck, 2001]. They are paraphrased below:

1. Place the highest priority on customer satisfaction through early and continuous delivery of valuable software systems.
2. Welcome changing requirements that enhance the customer's competitive, irrespective of the stage in the development.
3. Deliver working software frequently, and within a short timeframe.
4. Get the business people and the software developers to work together on a consistent basis throughout the project.
5. Build projects around motivated individuals. Give them the required resources and trust them to get the job done.
6. The most efficient and effective method of communication within a software engineering team is face-to-face conversation.

7. The primary measure of progress and success is a working software system or component.
8. The agile development process promotes sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. There should be continuous attention to technical excellence and good design.
10. There should be great emphasis on simplicity as an essential means of maximizing the amount of work not done.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. The software engineering team should periodically reflect on how to become more effective, and then refine its behavior accordingly.

The advantages of agile development are similar to those of phased prototyping:

- The user gets a version of the required software system in the shortest possible time.
- By merging business personnel and software developer in a single project, there is improved user participation.
- The likelihood of producing an acceptable system is enhanced.

Like the advantages, the disadvantages of the agile development model are comparable to those of the phased prototyping model:

- There is an increased likelihood of a poorly documented system. In fact, some extreme proponents of agile development go as far as to deemphasize the importance of software documentation.
- The system may be poorly integrated.
- A system that is poorly designed, documented, and poorly

integrated is likely to be more difficult to maintain, thus increasing the said cost that agile development seeks to control.

1.5 Categories of Software

Software engineering is a very wide, variegated field, constrained only by one's own imaginations and creativity. There are, however, some observable categories of software. [Figure 1-4](#) provides a list of common software categories. Most of the software products that you are likely to use or be involved with fall into one or more of these categories.

Operating System: A set of programs that provide certain desirable and necessary features for users of a computer system.
Compiler: A program that allows users (programmers) to code instructions to a computer system in a high level language (HLL). The compiler converts the instructions from source code to object (machine) code.
Interpreter: An interpreter is similar to a compiler. However, it operates in an interactive mode, whereas the compiler operates in batch mode.
Assembler: A special compiler that works on lower level (assembly language) programs, converting them to object code.
Database Management System (DBMS): A set of programs that facilitate the creation and management of a database. A database is a collection of related records. A database consists of at least one file containing data, but typically includes several files.
Network Protocol: A software system that facilitates electronic communication on a computer network, according to a prescribed set of rules and standards.
Desktop Applications: Describe all generic computer software applications that run on microcomputers and notebook computers. They include subcategories such as word games, multimedia applications, and web browsers.
Information System: A software system that facilitates the management of information. There are different kinds of information systems; these include batch processing systems, transaction processing systems, management information systems (MIS), decision support system (DSS), execute information systems (EIS), strategic information systems (SIS), expert systems (ES), hypermedia (documentation) systems, Web information system (WIS).
Note: A SIS provided support to the strategic management of an organization, a group of organizations, or an economy. DSS and EIS support decision-making by providing information on demand, drill down etc. A WIS is a Web-accessible information system.
Data Warehouse: An integrated, subject-oriented, time-variant, nonvolatile, consistent database, constructed from multiple source databases, and made available (in the form of read-only access) to support decision making in a business context.
Business Applications: Describe software applications that solve specific problems in a business. They include, but are not confined to desktop applications and some information systems. Business applications therefore include accounting packages, library management systems, manufacturing systems, desktop applications, college/University administration systems, inventory management systems, point of sale systems, airline reservation systems.
Artificial Intelligence (AI) Systems: A system that causes the computer to exhibit humanlike intelligence. Popular branches include neural networks, natural language processing and expert systems.
Expert System (ES): A special case AI system that emulates a human expert in a particular problem domain, e.g. medical diagnosis and robotics.
Hypermedia System: A special desktop application that facilitates the creation and maintenance of multi-media-based systems. This includes geographic information systems (GIS), documentaries, documentation systems, etc.
Computer Aided Design (CAD) System: Special business/desktop application used in manufacturing and architecture to design blueprints.
Computer Aided Manufacturing (CAM) System: Used in manufacturing environments.
Computer Integrated Manufacturing (CIM) System: A combination of CAD and CAM.
Computer Aided Software Engineering (CASE) Tool: A sophisticated software product that is used to automate design and construction of other software products.
Rapid Application Development (RAD) Tool: A brand of CASE tool that facilitates the rapid design and construction of other software applications.
Software Development Kit (SDK): A conglomeration of software products bundled together for the purpose of software development.

Figure 1-4. Common Software Categories

1.6 Alternate Software Acquisition

Approaches

Software may be acquired by any of the following approaches, each with its advantages and challenges:

- Traditional waterfall approach (in-house or via contracted work)
- Prototyping (phased or rapid, in-house or via contracted work)
- Iterative development (in-house or via contracted work)
- Assembly from re-usable components (in-house or via contracted work)
- Formal transformation (in-house or via contracted work)
- Agile development (in-house or via contracted work)
- Customizing an application software package
- End-user development
- Outsourcing

Whatever the acquisition approach that is employed, a software engineering objective of paramount importance is the production of software that has a significantly greater value than its cost. This is achieved by packing quality in the product from the outset — a principle that is emphasized throughout the text.

1.6.1 Discussion

As a personal exercise, determine what scenario(s) would warrant each approach.

1.7 Software Engineering Paradigms

There are two competing paradigms of software construction: the (traditional) *function-oriented* (FO) approach and the (more contemporary) *object-oriented*

(OO) approach. The two approaches, though sometimes divergent, are not mutually exclusive — an experienced software engineer can design and construct software systems using aspects of both approaches. This leads to a third alternative — the *hybrid* approach — which ideally borrows the strong points from both approaches, while avoiding the vulnerable points in either (see [Foster, 2010]).

This course pursues a balance between the object oriented paradigm and the function oriented paradigm, with a strong focus on the fundamentals, and an evident bias to the object-oriented paradigm. A full treatment of the object-oriented approach is (best) treated in another course — *object-oriented software engineering* (OOSE) or *object-oriented methodologies* (OOM). However, in keeping with the bias, the appendices contain useful additional information on OOM.

1.8 Desirable Features of Computer Software

The following are some desirable features of computer software:

- **Maintainability:** How easily maintained is the software? This will depend on the quality of the design as well as the documentation.
- **Documentation:** How well documented is the system?
- **Efficiency:** How efficiently are certain core operations carried out? Of importance are the response time and the system throughput.
- **User Friendliness:** How well designed is the user interface? How easy is it to learn and use the system?
- **Compatibility:** with existing software products.
- **Security:** Are there mechanisms to protect and promote confidentiality and proper access control?
- **Integrity:** What is the quality of data validation methods?
- **Reliability:** Will the software perform according to

requirements at all times?

- **Growth potential:** What is the storage capacity? What is the capacity for growth in data storage?
- **Functionality and Flexibility:** Does the software provide the essential functional features required for the problem domain? Are there alternate ways of doing things?
- **Differentiation:** What is unique about the software?
- **Adaptability:** How well are unanticipated situations handled?
- **Productivity:** How will productivity be affected by the software system?
- **Comprehensive Coverage:** Is the problem comprehensively and appropriately addressed?

These features are often referred to as *software quality factors* and for good reasons, since they affect the quality of the software products. In this course, you will learn how to research, plan, design, and construct software of a high quality. You will do so in a deliberately methodical manner. As such, we will revisit these quality factors later, and show how they can be used to guide the software engineering process.

1.9 Summary and Concluding Remarks

Let us summarize what has been covered in this chapter:

- A system is a set of interacting, interrelated, interdependent components that function together as a whole to achieve specific objectives.
- Software is the combination of program(s), database(s) and documentation in a systemic suite, with the sole purpose of solving specific system problems and meeting predetermined objectives.

- Software engineering is the process by which an information system or software is investigated, planned, modeled, developed, implemented and managed.
- The organization is a complex system consisting of personnel, facilities, equipments, materials and methods.
- Software systems are created specifically for organizational usage and benefits.
- Every software system has a life cycle — a period over which it is investigated/conceived, designed, development and remains applicable or needed. Various life cycle models have been proposed; this chapter examined the following seven models: waterfall, phased prototyping, iterative development, rapid prototyping, formal transformation, component-based development, and agile development.
- Irrespective of the life cycle model employed, computer software goes through the phases of investigation and analysis, design, development, implementation, and management. These phases are referred to as the software development life cycle (SDLC).
- Two paradigms of software construction are the function-oriented (FO) approach and the object-oriented (OO) approach.
- Among the desirable features of computer software are the following: maintainability, documentation, efficiency, user friendliness, compatibility, security, integrity, reliability, growth potential, functionality and flexibility, differentiation, adaptability, productivity, and comprehensive coverage.

So now you have an academic knowledge of what is meant by software engineering. As you will soon discover, this is not enough. To excel in this field, you will need experiential knowledge as well. But you have made an important start in exploring this exciting field. Much more could be said in this introductory chapter. If you are comfortable with the material presented and want to delve deeper into the OO paradigm for software engineering, please refer to [appendix 1](#), [appendix 2](#) and [appendix 3](#). The next chapter will discuss the

role of the software engineer in the organization.

1.10 Review Questions

1. Give definitions of:
 - A system
 - Computer software
 - Software engineering
2. Explain how software engineering relates to the management of an organization.
3. Develop an organization chart for an organization that you are familiar with. Analyze the chart and propose a set of interrelated software systems that may be used in helping the organization to achieve its objectives.
4. Why should a software engineer be cognizant of the information levels in an organization?
5. What is the software development life cycle? Explain its significance.
6. Discuss the seven life cycle models covered in the chapter. For each model:
 - Describe the basic approach
 - Identify the advantages
 - Identify the disadvantages
 - Describe a scenario that would warrant the use of this approach
7. Identify ten major categories of computer software. For each category, identify a scenario that would warrant application of such a category of software.
8. Discuss some desirable features of computer software.
9. Write a paper on the importance of software engineering, and your expectations for the future.

1.11 References and/or Recommended Readings

[Beck, 2001] Beck, Kent, et. al. *Manifesto for Agile Software Development*. 2001. <http://www.agilemanifesto.org/> (accessed June 2009).

[Bruegge, 2004] Bruegge, Bernd and Allen H. Dutoit. *Object-Oriented Software Engineering* 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2004. See [chapter 1](#).

[Foster, 2010] Foster, Elvis C. with Shripad Godbole. *Database Systems: A Pragmatic Approach*. Bloomington, IN: Xlibris Publishing, 2010. See chapter 23.

[Harris, 1995] Harris, David. *Systems Analysis and Design: A Project Approach*. Fort Worth, TX: Dryden Press, 1995. See [chapters 1](#) and [2](#).

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [chapters 1](#) and [2](#).

[Loudon, 1994] Laudon, Kenneth C. and Jane P. Laudon. *Management Information Systems* 4th ed. Eaglewood Cliffs, NJ: Prentice Hall, 1994. See [chapters 1 – 4, 12, 15, 16, 17, 18](#).

[Long, 1989] Long, Larry. *Management Information Systems*. Eaglewood Cliffs, NJ: Prentice Hall, 1989. See [chapters 1–3](#).

[Maciaszek, 2005] Maciaszek, Leszek A. and Bruce Lee Long. *Practical Software Engineering*. Boston, MA: Addison-Wesley, 2005. See [chapter 1](#).

[Peters, 2000] Peters, James F. and Witold Pedrycz. *Software Engineering: An Engineering Approach*. New York, NY: John Wiley & Sons, 2000. See [chapters 1](#) and [2](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapters 1](#) and [2](#).

[Pressman, 2005] Pressman, Roger. *Software Engineering: A Practitioner's Approach* 6th ed. Crawfordsville, IN: McGraw-Hill, 2005. See [chapters 1–3](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [chapters 1–3](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See [chapters 1-3](#).

[Spence, 2004] Spence, Ian and Kurt Bittner. Managing Iterative Software Development with Use Cases. IBM, June 2004.

<http://www.ibm.com/developerworks/rational/library/509>
(accessed December 2009).

[Van Vliet, 2000] Van Vliet, Hans. *Software Engineering* 2nd ed. New York, NY: John Wiley & Sons, 2000. See [chapters 1](#) and [2](#).

CHAPTER 2



The Role of the Software Engineer

This second chapter will examine the role, responsibilities, functions and characteristics of the software engineer in the organization. The following will be discussed:

- Historical Role
- Modern Role of the Software Engineer
- Job Description of the Software Engineer
- Tools Used by the Software Engineer
- Management Issues With Which the Software Engineer Must be Familiar
- Summary and Concluding Remarks

2.1 Historical Role

Historically, most systems were manual and coordinated by a systems and procedure analyst with responsibilities such as:

- Forms design and record management
- Analysis, design and management of manual systems
- Report distribution analysis
- Work measurement and specification
- Development and maintenance of procedure manuals

The systems and procedure analyst was usually attached to the Finance Department. Most systems were accounting oriented. This created an imbalance. A second difficulty was the collation of information from various departments.

The arrival of computer technology triggered several changes and brought several advantages:

- A vast amount of data could be collected and stored.
- Inherent validation checks could ensure data accuracy.
- The response was fast.
- There was significant reduction of manual effort, redundancies and data security problems.

2.2 Modern Role of the Software Engineer

In addition to the historical roles, the software engineer (SE) takes on additional responsibilities, some of which are mentioned here:

- The software engineer acts as a *consultant* to the organization.
- The software engineer is the *supporting expert* on system design, development, implementation and maintenance.
- He/She is the *change agent* in the organization, lobbying for and effecting system enhancements (manual and automated; strategic and functional).
- The software engineer acts as a *project leader* on the development or modification of systems.

The software engineer acts as a *software designer and developer* (especially in developing countries or other environments where resources are scarce).

The contemporary trend reflects a bias to the title of software engineer instead of the traditional systems analyst (SA). Software engineer is more accurate as it spans the whole software development life cycle. However, many organizations tend to stick to the traditional. Where both titles are used in the

same organization, the SA usually operates upper phases of SDLC, while the software engineer operates in the lower phases; some overlap typically takes place in the area of software design.

2.3 Job Description of the Software Engineer

If you are interviewed for, or employed as a software engineer, one of the first pieces of document you will receive is a job description. Each organization has its own standards as to how job descriptions are written. However, generally speaking, the job description will have the following salient components:

- **Heading:** An organizational heading followed by a departmental heading.
- **Summary:** A summary of what the job entails.
- **Core Functions:** A list of core functions and/or responsibilities of the job. This is further elucidated below.
- **Desirable Qualities:** A section describing the desirable qualities of the incumbent. This is further elucidated below.
- **Authority:** A section that outlines the authority associated with the job.
- **Job Specification:** A section that describes reporting relationship and other related information about the job.

[Figure 2-1](#) provides an illustration of a job description of a software engineer. It includes all the essential ingredients mentioned above. The next two subsections further provide content guidelines for the core functions of a software engineer, and the desirable qualities of the incumbent.

LAMBERT COX COLLEGE JOB DESCRIPTION
Department of Computer Science

JOB TITLE of Software Engineer

SUMMARY

The incumbent works with the software engineering team of the college to develop and maintain quality software systems that will be relevant to the college in particular and the information technology industry in general. The incumbent also participates in academic matters as the need arises.

CORE FUNCTIONS

01. Investigates and determines the information systems needs of user departments as assigned by the Chair of Computer Science Department.
02. Prepares detailed requirements specification and design software systems to be developed.
03. Discusses and confirms the project concept with Director of Computer Science and Director of Computer Services.
04. Develops design specifications of various software systems for review and approval.
05. Participates in the establishment of software development standards for the institution.
06. Develops software systems according to approved and agreed specifications and standards.
07. Maintains software systems previously developed as required.
08. Prepares and submits relevant reports and documentation on systems developed.
09. Conducts training of end users of various information systems and software, lab assistants and other members of staff as required.
10. Participates in consulting ventures on behalf of the college.
11. Participates in teaching of no more than one course per quarter if and when required.
12. Keeps abreast of current developments and trends in information technology and ensures that the college is likewise kept up-to-date.
13. Provides leadership and guidance to students and/or trainees who may from time to time work in the Software Engineering Center.
14. Performs any other related duties consistent with the nature, functions and objectives of the job.

AUTHORITY TO

01. Participate in budget preparation and review exercises.
02. Design and develop resource materials.
03. Take necessary action to ensure security and proper use of resources.
04. Make recommendations for the selection of resources.
05. Apply disciplinary measures to students and/or trainees working in the Software Engineering Center.

JOB SPECIFICATION

Required Qualification

Post graduate degree in computer science or related field

Alternate Qualification

- Bachelor of science in computer science or related field
- Bachelor of science (or graduate diploma) in management studies

Required Experience

A minimum of five years in systems development, implementation and management is required. In the absence of such experience,
the incumbent will be subject to one year of intense on-the-job training and guidance.

Required Expertise

- Excellent communication skills
- Conversant with various computer platforms, information systems methodologies and approaches
- Excellent human resource management skills
- Excellent software development skills

Special Requirement

- Flexible working hours
- Sometimes required to work outside of normal working hours

Reports to Chairman, Computer Science Department

Liaises with Chair of Computer Services, employees at all levels of operation of the college, external organizations in the field of information technology, and other companies as required.

Figure 2-1. Sample Job Description for a Software Engineer

2.3.1 Core Functions of the Software Engineer

Some core functions of the software engineer are as follows:

- Investigates and defines software system problems, and makes proposals for their solution.
- Carries out detailed feasibility analysis and system specification.
- Plays a critical role in the design and development of software systems in the organization.
- Participates in the process of hardware/software selection for the organization.
- Participates in the definition of software engineering standards for the organization.
- Performs as project leader according to assignment from his superior.
- Conducts system enhancements as required.
- Keeps abreast of current technological developments and trends, and ensures that the organization is strategically positioned.
- Plays a crucial role in the development and maintenance of system and user documentation.
- Plays a crucial role in the training of staff.
- Ensures the safety of all information technology (IT) related resources.

2.3.2 Desirable Qualities of the Software Engineer

Among the required qualities of the software engineer are the following:

- Excellent communication skills
- Pleasant personality
- Tolerant and businesslike
- Experienced in various computer platforms, applications and software
- Wide experience in (appreciation of) business administration
- Able to effectively and efficiently assimilate large volumes of information
- Imaginative and perceptive
- Able to understand complex problems
- A good sales person of ideas

2.4 Tools used by the Software Engineer

Traditionally, the tools the software engineer uses in performing include:

- Coding Systems
- Forms Design
- Data Analysis Charts
- Decision Tables and Decision Trees
- Flow Charts and Diagrams
- Other Creative Technical Documents and Modeling Techniques
- Software Planning and Development Tools

Additionally, *object-oriented software engineering* (OOSE), also referred to as *object-oriented methodologies* (OOM), provides other standards, conventions and tools. This course covers the main OO design techniques and

methodologies; however a more detailed treatment can be obtained from a course in OOSE or OOM (see the appendices). Flow charts, diagrams and decision tables/trees are covered in lectures 5 and 6 respectively. The other mentioned tools will be briefly discussed here.

2.4.1 Coding Systems

A code is a group of characters used to identify an item of data and show its relationship to other similar items. It is a compact way of defining a specific item or entity. Coding systems are particularly useful during forms design and database design (to be discussed in [chapter 10](#)). Benefits of a proper coding system include:

- Reduction of storage information
- Easy identification and recollection
- Easy classification of data

Desirable Features of a Coding System

A coding system must exhibit some of the following features:

- **Uniqueness:** The code must give specificity to different items.
- **Purpose:** The code must serve a useful purpose (e.g. comparison or location).
- **Compactness:** The code must not be too bulky.
- **Meaningful:** The code must be meaningful. The code can relate to shape, size, location, type, or other features of items represented.
- **Self-checking:** The code may be self-checking, utilizing a mathematically calculated check digit as part of the code to ensure validity.
- **Expandable:** The code must be expandable; it must be easy to add new blocks of items.

Types of Coding Systems

There are six common types of coding systems as summarized below. Bear in mind however, that in many practical cases, coding types are combined to form the desired coding system.

1. Simple Sequence

- Numbers are assigned consecutively, e.g. 1, 2, 3, ...
- The main advantage of the approach is that an unlimited number of items can be stored.
- The main drawback is that little or no information conveyed about items.

2. Block Sequence

- Blocks of consecutive numbers identify groups of similar items.
- The main advantage of this approach is that more information is conveyed about the items.
- The main drawback is that it is not always possible to maintain sequencing in each block.

3. Group Classification

- This approach involves a major, intermediate and minor classification of items by ordering or digits. A good example is the US zip code.
- The main advantage of the approach is that data can be subdivided.
- One possible drawback is that groups can become quite large.

4. Significant Digit

- For this type of code, a digit is assigned to denote a physical characteristic of the item.
- The main advantage of the approach is that it aids physical identification of the items.

- Except for a few simple scenarios, the approach is inadequate to be used on its own. For this reason, it is often combined with another type of coding.

5. Alphabetic Codes

- This coding system involves the use of letters to identify items. The coding system may be mnemonic (e.g. H₂O represents water) or alphabetic according to a set rule (e.g. NY represents New York).
- The main advantage of the approach is that it facilitates quick reference to items.
- One drawback is that if not controlled, the code used can be too long and confusing.

6. Self-checking Code

- In this coding system, a check digit (or number) is mathematically calculated and applied to the code.
- This approach is particularly useful in situations where data validation is required after transmission over telecommunication channels.
- The approach is redundant in situations where electronic transmission is not required.

2.4.2 Forms Design

A significant proportion of data input to a business information system will be done with the use of forms. For instance, most universities and colleges require applicants to fill out application forms. Completed application forms are then used to key data into some database for subsequent processing. When an applicant is accepted, it is this same data from the application form that is used as the student's personal information.

Forms design provides the following advantages:

- Elimination redundant operations
- Reduction of the organization's operational cost

- Facilitation of filing and retrieval
- Enabling of easy data capture
- Facilitation of interdepartmental flow
- Facilitation of control e.g. assigning a form number or color code

A form has three parts: a heading, a body, and a footing. The heading should contain the important company information (for example the company name), along with a descriptive title for the form. The body contains the essential details that the form contains. The footing contains instructional guidelines, and is optional (some forms are self-explanatory and therefore do not need footnotes). [Figure 2-2](#) indicates the three styles for providing/requesting information on a form – open style, box style, or a mixture of both.

Fill-in Style	Example
Open Styles	Name: _____ Address: _____
Box Style	Name: <input type="text"/> <input type="text"/> Address: <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>
Mixture of both Styles	

[Figure 2-2](#). Form Fill-in Styles Used

There are three broad categories of forms, as described below:

1. **Cut Sheet Forms:** These are usually made in pads; sheets can be detached. This is the most common type of form.
2. **Specialty Forms:** There are four basic types of specialty forms:
 - Bound in books e.g. receipts

- Detachable stub e.g. check
 - Mailer
 - Continuous e.g. machine printed report
3. **Electronic Forms:** These are forms for which there may or may not be printed versions. They serve as either data collection instruments for system input, or information dissemination instruments for system output.

Guiding Principles of Forms Design

In designing forms, the following guiding principles are important:

- Captions must be unambiguous; instructions must be clear, copies must be numbered or color-coded and/or numbered.
- The form should require minimum writing by the person filling it in, collecting only necessary data.
- Arrangement of information must be logical and subservient to efficiency.
- The form must be easy to fill out.
- The completed form should be easy to use in the system.
- The form layout must be consistent with input screen design (see [chapter 11](#)).
- The form must have good legibility features. In support of legibility, the Le Courier's legibility chart is famous ([Figure 2-3](#)).

Rank	Print	Background
1	Black	Yellow
2	Green	White
3	Red	White
4	Blue	White
5	White	Blue
6	Black	White
7	Yellow	Black
8	White	Red
9	White	Green
10	White	Black
11	Red	Yellow
12	Green	Red
13	Red	Green

Figure 2-3. Le Courier's Legibility Chart

2.4.3 Data Analysis Charts

Data analysis charts are used to present information in a clear, concise manner. Uses of charts include:

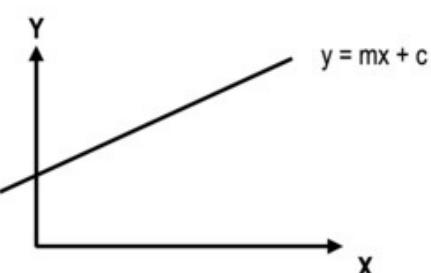
- Analysis of trends
- Forecasting of future events
- Comparison of data
- Conveying technical information about the requirements of a system

Four commonly used types of charts (illustrated in [Figure 2-4](#)) are:

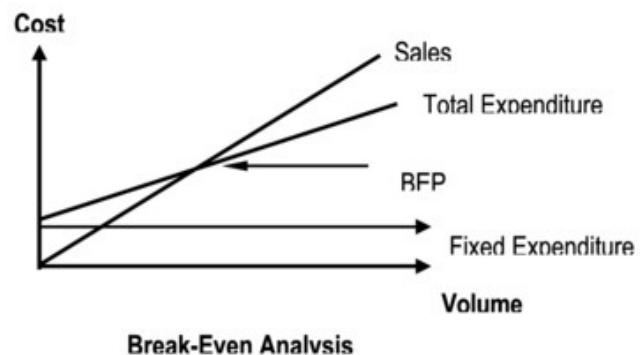
- Line Chart, as used in regression analysis and break-even analysis
- Pie Chart, used to show percentage distributions
- Bar Chart, used to show movements and relationships
- Step Chart, as in a histogram

Note: Since this chart was proposed, computer graphics has significantly improved; today, the list of possible colors is much richer. The chart should therefore be construed as a basic guideline.

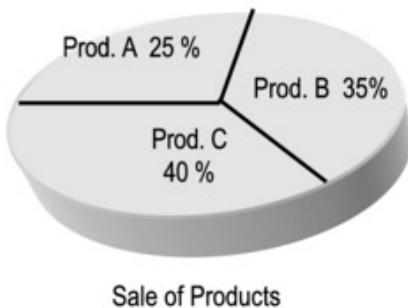
a. Line Charts:



Regression Line



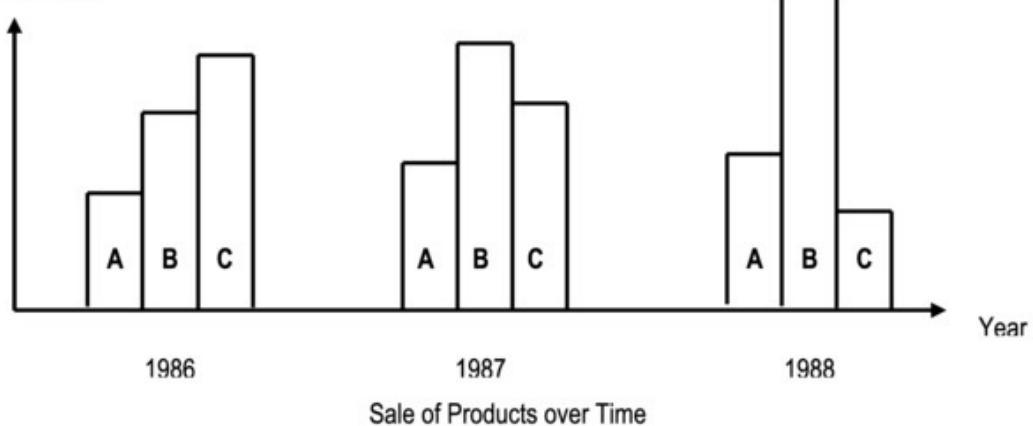
b. Pie Chart:



Sale of Products

c. Bar Chart:

Sale (1,000)



Sale of Products over Time

d. Histogram:

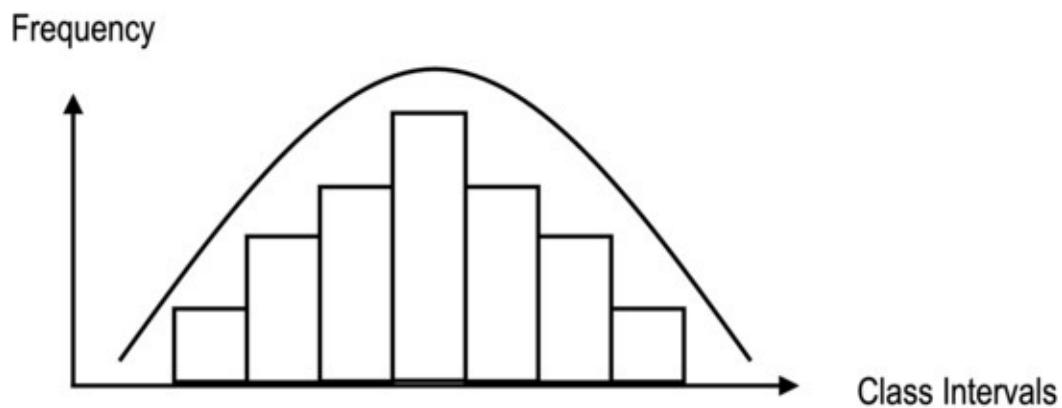


Figure 2-4. Examples of Various Charts

2.4.4 Technical Documents and Modeling Techniques

In carrying the responsibilities of the job, the software engineer prepares several technical documents. There are no set rules or formats for these; the software engineer may be as innovative as he/she wishes provided that the following guidelines are observed:

- The document must be concise, but comprehensive in its treatment of the related subject.
- Where relevant, established tools and techniques must be employed.

The following are some examples of technical documents (typically referred to as *deliverables*) that the software engineer might be responsible for preparing:

- Project Proposal
- Initial System Requirement
- Feasibility Report
- Requirements Specification

- Design Specification
- Help System
- User's Guide
- System Manuals

With respect to modeling techniques, this course, along with a course in OOM, will cover most of the established techniques that are at the disposal of the software engineer. Additionally, the experienced software engineer will from time to time, devise innovative variations of standard techniques, commensurate with the problem domain under consideration. He/she may even propose new techniques. To think about it for a moment, all the methodologies and techniques that are covered in this course are approaches that were developed by software engineers, in their attempts to solve problems.

2.4.5 Software Planning and Development Tools

Software planning and development tools were mentioned in the previous chapter. They may be loosely classified as modeling and design tools, database planning tools, software development tools, and DBMS suites. Some of the popular products have been included in [Figure 2-5](#) (in alphabetic order). Please note that this is by no means a comprehensive list. As you view it, please note the following:

- Software engineering is a challenging, exciting, and rewarding field that is supported by a rich reservoir of excellent resources to help you learn and gain mastery in the discipline.
- Terms such as *CASE tool* and *RAD tool* describe a wide range of software systems that include any combination of the above-mentioned classifications (for more on this, see [appendix 7](#)). One's choice of CASE/RAD tool(s) will depend on the prevailing circumstances and the desired objectives for the software engineering project.

Product	Parent Company	Comment
Software Model and Design Tools		
ConceptDraw	CS Odessa	Supports UML diagrams, GUI designs, flowcharts, ERD, and project planning charts.
Enterprise Architect	Sparx Systems	Facilitates UML diagrams that support the entire software development life cycle (SDLC). Includes support of business modeling, systems engineering, and enterprise architecture. Supports reverse engineering as well.
MagicDraw	No Magic	A relatively new product that has just been introduced to the market. Appears to be similar to Enterprise Architect.
Power Designer	Sybase	Supports UML, business process modeling, and data modeling. Integrates with development tools such as .NET, Power Builder (a Sybase Product), Java, and Eclipse. Also integrates with the major DBMS suites.
SmartDraw	SmartDraw	A graphics software that facilitates modeling in the related disciplines of business enterprise planning, software engineering, database modeling, and information engineering (IE). Provides over 100 different templates (based on different methodologies) that you can choose from. Supported methodologies include UML, Chen Notation IE notation, etc.
TogetherSoft	Borland	Provides UML-based visual modeling for various aspects of the software development life cycle (SDLC). Allows generation of DDL scripts from the data model. Also supports forward and reverse engineering for Java and C++ code.
Toolkit for Conceptual Modeling (TCM)	University of Twente, (of Holland)	Includes various resources for traditional software engineering methodologies as well as object-oriented methodologies based on the UML standards.
UML Diagrammer	Pacestar	Facilitates easy creation of UML diagrams for software systems.
Visio	Microsoft	Facilitates modeling in support of business enterprise planning, software engineering, and database management.
Visual Thought	CERN	Similar to Visio but is free

Product	Parent Company	Comment
Database Model and Design Tools		
DataArchitect	theKompany.com	Supports logical and physical data modeling. Interfaces with ODBC and DBMSs such as MySQL, PostgreSQL, DB2, MS SQL Server, Gupta SQLBase, and Oracle. Runs on Linux, Windows, Mac OS X, HP-UX, and Sparc Solaris platforms.
Database Design Studio	Chilli Source	Allows modeling via ERD, data structure diagrams, and data definition language (DDL) scripts. Three products are marketed: DDS-Pro is ideal for large databases; DDD-Lite is recommended for small and medium-sized databases; SQL-Console is a GUI-based tool that connects with any database that supports ODBC.
Database Design Tool (DDT)	Open Source	A basic tool that allows database modeling that can import or export SQL.
DBDesigner 4 and MySQL Workbench	fabFORCE.net	This original product was developed for the MySQL database. The replacement version, MySQL Workspace is targeted for any database environment, and is currently available for the Windows and Linux platforms.
DeZign	Datanamic	Facilitates easy development of ERDs and generation of corresponding SQL code. Supports DBMSs including Oracle, MS SQL Server, MySQL, IBM DB2, Firebird, InterBase, MS Access, PostgreSQL, Paradox, dBase, Pervasive, Informix, Clipper, Foxpro, Sybase, SQLite, ElevateDB, NexusDB, DBISAM.
ER Creator	Model Creator Software	Allows for the creation of ERDs, and the generation of SQL and the generation of corresponding DDL scripts. Also facilitates reverse engineering from databases that support ODBC.
ER Diagrammer	Embarcadero	Similar to ER Creator
ERWin Data Modeler	Computer Associates	Facilitates creation and maintenance of data structures for databases, data warehouses, and enterprise data resources. Runs on the Windows platform. Compatible with heterogeneous DBMSs
Oracle Designer	Oracle	Supports design for Oracle databases.
Oracle JDeveloper	Oracle	Supports UML diagramming.
xCase	Resolution Software	A database modeling tool that supports all aspects of the database development life cycle (DDLC): it supports ERD design, documentation, SQL code generation, logical and physical migration across multiple DBMS platforms, and data analysis.

Product	Parent Company	Comment
Software Development Tools		
Delphi, C++ Builder, JBuilder, and Kylix	Formerly by Borland; Now marketed by Embarcadero Technologies	A group of RAD tools that facilitates the development of various categories of software systems (including database connectivity). Code generated in Object Pascal, C++, or Java (depending on user's choice). Runs on various operating system (OS) platforms.
LiveModel	Intelllicorp	An OO-CASE tool that facilitates modeling and development of various software systems.
LANSA	LANSA Corporation (of Australia)	An integrated CASE tool for OS/400 and Windows environments.
MS Visual Studio	Microsoft	A conglomerate of software development environments including Visual Basic, Visual C++ and Visual FoxPro
NetBeans	Sun Microsystems	A Java development environment that runs on various OS platforms.
Object Domain Standard	Object Domain Systems	Facilitates forward and reverse engineering for C++, Java, Python, SVG, and Python Scripting.
Rational Rose	IBM	An OO-CASE tool for various OS platforms including UNIX and Windows
SAP	SAP AG (of Germany)	A suite of products that are used in <i>enterprise resource planning</i> (ERP). The English equivalent of the German acronym SAP means Systems, Applications and Products.
Software through Pictures UML	Aonix	Facilitates forward and reverse engineering for C, C++, Java, and Ada.
Synon	Computer Associates	A CASE tool for OS/400 and Unix environments.
Team Developer	Formerly by Gupta Technologies; now marketed by Unify Corporation	An OO-CASE tool that facilitates the development of various categories of software systems (including database connectivity) on the Windows platform.
UML Studio	Pragsoft Corporation	Facilitates forward and reverse engineering for C++, Java, and IDL.
Visual UML Standard Edition	Visual Object Modelers	Facilitates forward and reverse engineering for C++, C#, and Java. Also supports data modeling.
Database Management Systems (DBMSs)		
DB2	IBM	A universal DBMS (UDBMS) suite for various OS platforms including OS-400, Windows, UNIX and RISC.
Informix	IBM	A DBMS suite for various OS platforms including UNIX, Linux, Windows and RISC.
MySQL	Formerly MySQL AB (of Sweden); now Marketed by Oracle	An open source DBMS that is gaining in credibility. Runs on major OS platforms including Windows, Linux, and UNIX.
Oracle	Oracle	A universal DBMS (UDBMS) suite for various OS platforms, including UNIX, Linux, and Windows.
SQL Server	Microsoft	A universal DBMS suite for the Windows platform.
Sybase	Sybase	A DBMS suite for various OS platforms including UNIX, Windows, and Solaris.

Figure 2-5. Some Popular Software Planning and Development Tools

Given such a wide plethora of software engineering tools and resources, the

Given such a wide plethora of software engineering tools and resources, the matter of software evaluation and selection becomes pertinent to the success of a software engineering venture. Evaluation and selection should be conducted based on clearly defined criteria. This process will be further discussed in the upcoming chapter.

You have no doubt noticed that many of the tools mentioned in [figure 2-5](#) relate to the field of database systems. This happens to be a very important field of software engineering that will be summarized in [chapter 10](#), but which you will (or should) study in a separate course. A database management system (DBMS) is the software that is used to develop and manage a database. There are two main categories of DBMS suites (see [Date, 2004] and [Foster, 2010]):

- **Relational:** A relational DBMS supports the relational model for database design.
- **Universal:** A universal DBMS supports the relational model as well as the object-oriented model for database design.

CASE tools may be placed into three broad categories, each with three sub-categories [Martin, 1993]:

- **Traditional:** A traditional CASE tool is CASE tool that is based on the (traditional) function-oriented approach to software development. It may be a front-end system (concentrated on software investigation, analysis and specification), a back-end system (concentrated on software development), or an integrated (concentrated on the entire SDLC).
- **Object-oriented:** An OO-CASE tool is a CASE tool that is based on the OO paradigm for software construction. It may be front-ended, back-ended, or integrated.
- **Hybrid:** A hybrid CASE tool is a CASE tool that supports both the FO paradigm and the OO paradigm for software construction. It may be front-ended, back-ended, or integrated.

Most RAD tools tend to be object-oriented. Moreover, the distinction between CASE and RAD is not always well defined. However, what is not in question is spectrum of significant benefits that these software development tools bring to the software engineering discipline. These benefits include (but are

(not necessarily confined to):

- Automated project management techniques (see chapter 8)
- System modeling (including database and user interface)
- Conveniences of graphical user interface
- Prototype modeling and testing
- Sophisticated diagramming techniques, including executable diagrams
- Fourth generation language (4GL) interface to aid in application development
- Automatic code generation

2.5 Management Issues with Which the Software Engineer must be Familiar

Whether your organization is a software engineering firm, or a typical business that specializes in some (traditional or non-traditional) field of interest, there is a virtual inevitability that it will need the services of software engineers. Once the enterprise relies on, uses, or markets software, such services will be required. Software engineering has therefore become a pervasive, ubiquitous discipline.

The software engineer must be acquainted with plans that will significantly affect the organization. The effects may be in any of a number of areas, for instance:

- Plans for expansion, or contraction, or reorganization might affect the physical infrastructure of the organization.
- Relocation will affect the physical infrastructure also.
- A merger or contraction will affect the scope of the organization's information infrastructure.
- Changes in the methods of internal work (e.g. budget

preparation) or mode of operation (global or in specific areas), might require adjustment to underlying information system operations.

- Changes in the business focus of the enterprise may also impact on organization's information infrastructure.

In many cases, these changes directly impact the software systems on which the organization relies; in other instances, the effect is indirect. Whether the effects are direct or indirect, the underlying systems may require modification, and the software engineering team will be required to implement these modifications.

2.6 Summary and Concluding Remarks

Here is a summary of what we have covered in this chapter:

- The software engineer is responsible for the investigation, analysis, design, construction, implementation and management of software systems in the organization. This requires a highly skilled individual with certain desirable qualities.
- The software engineer uses a number of important tools in carrying out the functions of the job. These include coding systems, forms design, charts, diagrams, technical documents, and software development tools.
- Coding systems allow for data representation, particularly during database design and forms design.
- Forms design relates to data collection instruments for input to as well as output from the system.
- Data analysis charts are used for data representation in a concise, unambiguous manner. Line charts, pie charts, bar charts and step charts are commonly used.
- Technical documents convey useful information about the

work of the software engineer. These include but are not confined to project proposal, initial system requirement, feasibility report, requirements specification, design specification, help specification, user's guide, and system manual. We will discuss these documents as we proceed through the course.

- Flow charts and other diagrams will be discussed later in the course.
- Software planning and development tools are available in the form CASE/RAD tools and SDKs in categories such as model and design tools, database planning tools, software development tools, and DBMS suites.
- The software engineer must be aware of any planned organizational changes that will have direct or indirect implications for the software systems of the organization.

This completes the introductory comments about software engineering. If you are comfortable with the material presented and want to delve deeper into the OO paradigm for software engineering, please refer to [appendix 1](#), [appendix 2](#) and [appendix 3](#). The next few chapters will discuss various issues relating to software requirements investigation and analysis.

2.7 Review Questions

1. Prepare a job description for a software engineer in a software engineering firm.
2. Propose coding systems for the following:
 - Animals in a zoo
 - Employees in an organization
 - Students at a college
 - Resources in a college library
 - Airline flights at an international airport
3. Imagine that you were employed as a software engineer

for a firm. Your team is in the process of overhauling the company's information infrastructure. You have been asked to propose a front-end software application that will automate the job application process. Do the following:

- Propose a job application form for the company.
 - Identify the steps that you envisage that a job application would pass through.
 - After an applicant has been employed, what (internal) software system(s) do you suppose your application processing system would feed its output into? Why?
4. Give examples of technical documents that software engineers have to prepare in the course of their job.
 5. Give examples of some contemporary software planning and development tools that are available in the marketplace.
 6. Why do software engineers need to concern themselves with management issues in an organization?

2.8 References and/or Recommended Readings

[CDP, 2006] CDP Print Management. Le Couriers Table of Legibility. 2006. <http://www.cdp.co.uk/lecourierstable.shtml> (accessed July 2009).

[Date, 2004] Date, Christopher J. *An Introduction to Database Systems* 8th ed. Reading, MA: Addison-Wesley, 2004.

[Foster, 2010] Foster, Elvis C. with Shripad Godbole. *Database Systems: A Pragmatic Approach*. Bloomington, IN: Xlibris Publishing, 2010.

[Martin, 1993] Martin, James. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1993.

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapter 2](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [chapter 5](#).

PART B



Software Investigation and Analysis

The next six chapters will focus on the Investigation and Analysis Phase of the SDLC. The objectives of this phase are

- To clearly identify, define and describe software system problems
- To thoroughly analyze system problems
- To identify and evaluate alternate solutions to identified software system problems. Such solutions should improve efficiency and productivity of the system(s) concerned
- To provide adequate documentation of the requirements of software systems.

Two deliverables will emerge from this phase of the software development life cycle (SDLC): the initial software/system requirement (ISR) and the requirements specification (RS).

Chapters to be covered include:

- [Chapter 3](#) — Project Selection and Initial System Requirement
- [Chapter 4](#) — Requirements Specification
- [Chapter 5](#) —Information Gathering
- [Chapter 6](#) — Communication Via Diagrams
- [Chapter 7](#) — Decision Models For System Logic

- [Chapter 8](#) — Project Management Aid

CHAPTER 3



Project Selection and the Initial System Requirements

This chapter covers a number of activities that are typical of the early stage of software requirements investigation. These activities converge into the first deliverable — the *initial system requirement* (ISR), also called the *project proposal*. The ISR contains a number of important components, which will be discussed. The chapter proceeds under the following subheadings:

- Project Selection
- Problem Definition
- Proposed Solution
- Scope and Objectives of the System
- System Justification
- Feasibility Analysis Report
- Alternate Approach to Feasibility Analysis
- Summary of System Inputs and Outputs
- Initial Project Schedule
- Project Team
- Summary and Concluding Remarks

3.1 Project Selection

For obvious reasons, let us assume that an organization stands in need of various software systems. The directors of the organization may or may not know the intricate details of their need, but hopefully are smart enough to recognize that something is lacking. At this early stage, management would invite the attention of one or more IT and/or software systems professionals (from the host organization, or a contracted software engineering firm) to conduct a needs assessment and make appropriate recommendations.

Very early in this software needs assessment exercise, the matter of project selection would become pertinent: It is often common that an organization stands in need of several software systems. Since it is impractical to attempt to acquire them all at the same time, some determination is normally made as to which projects will be pursued first. The intricacies of such determination are beyond the scope of this course; however, you should be aware of the salient factors that will likely affect such decisions:

- Backing from management
- Appropriate timing
- Alignment to corporate goals
- Required resources and related cost
- Internal constraints of the organization
- External constraints of the environment/society

If the organization in question is a software engineering firm, there are some additional factors to consider:

- Anticipated marketability of the product
- Expected impact on the industry
- Expected market position to be attained by introducing this new software

Consideration of these factors against each prospective software engineering project will lead to selection on one or more projects to be pursued. Once a project is selected, the next target is preparation of the ISR. The rest of this chapter discusses various components of the ISR, starting with the problem definition.

3.2 Problem Definition

Once the software engineering project has been selected, the first activity that is performed is that of problem definition. It must therefore be done thoroughly, accurately, and persuasively. The following are some basic guidelines:

- Define what aspect(s) of the organization is affected (recall the functional areas of an organization).
- Describe (in convincing, precise language) the effects and possible causes. Show how problems may be interrelated.
- Describe the worker attitude to the problem.
- Cite the impact the problem has on productivity and revenue.
- Identify the operating constraints.

3.2.1 Constraints of a System

In defining a system, there are certain constraints that the software engineer must be aware of. These include the following:

- External Constraints
 - Customers of the organization
 - Government policies
 - Suppliers of the organization
 - State of the economy
 - State of the industry in which the organization operates
 - Competitors of the organization
- Internal constraints include
 - Management support
 - Organizational policies
 - Human resource

- Financial constraints
- Employee acceptance
- Information requirements
- Organizational politics

3.2.2 Aid in Identifying System Problems

How does the software engineer identify system problems? The following pointers should prove useful:

1. Check output against performance criteria. Look for the following:
 - a. Too many errors
 - b. Work completed slowly
 - c. Work done incorrectly
 - d. Incomplete work
2. Observe behavior of employees. Look for the following:
 - a. High absenteeism
 - b. High job dissatisfaction
 - c. High job turnover
3. Examine source documents. Look for the following:
 - a. Inadequate forms e.g. poor layout
 - b. Redundant information on forms
 - c. Information required but not included on forms
4. Examine the work process in the organization. Look for the following:
 - a. Inadequate/poor work management
 - b. Redundancies
 - c. Omissions

5. Listen to external feedback from vendors, customers, and suppliers. Look for the following:
 - a. Complaints
 - b. Suggestions for improvement
 - c. Loss of business

3.2.3 Identifying the System Void

If the organization in question is a software engineering firm, or an organization that lacks adequate software systems support, then what might be the issue is not necessarily identifying problems in an existing system, but identifying the need for introducing a new software system.

If the proposal is for a new software system for internal use, the proposal should follow a thorough internal research that involves prospective users of the product. The research should clearly outline the areas of user complaint. A few points to note:

- The research might be conducted formally, involving the use of questionnaires (see [chapter 5](#)), user forums/workshops, expert review, and other information collection instruments.
- Information could also be gathered via informal/non-conventional means, for instance brain-storming and observation (see [chapter 5](#)).

If the proposal is for a new software system to be marketed to the public, identification of the need could arise from any combination of a brainstorming session, a market research, or industry observation.

3.3 The Proposed Solution

The software engineer must propose a well thought out solution to the problem as outlined. This typically includes a system name, operating platform, and required resources. Following are a few points about the proposed solution:

- The proposed solution may be to modify the existing software system or to replace it.
- The proposal must show clearly how the problems specified will be addressed.
- It should briefly mention what areas of the organization will be impacted and how.
- The proposal must mention the chosen system platform (do not include details here as this will be required in the feasibility report).
- The proposal should mention the estimated cost of the software system and project duration.

3.4 Scope and Objectives of the System

In specifying the scope of the proposed solution, the software engineer must be guided by the following:

- Identify main components (subsystems and/or modules) to your best knowledge (which is limited at this early stage).
- Identify the principal user(s) of the system.
- Cite the organizational areas to be impacted by the system.
- State for each (functional) area, whether user participation will be data input, processing of data or access to data.

An itemized list of objectives serves to sharpen focus on the essentials of the system. Basic guidelines for stating system objectives are:

- State system objectives clearly and concisely.
- Identify primary objectives and secondary objectives where applicable.
- Ensure that the needs of end-users are addressed, especially

those of the principal users.

- Align system objectives to corporate goals.
- Use corporate language in stating the objectives.

3.5 System Justification

It is imperative that the proposal underscores the benefits that the new software system brings to the table. These benefits go a considerable way to justifying the project in the eyes of management. Basic guidelines for stating software system benefits are as follows:

- Show how the system will help to improve productivity and efficiency.
- Mention advantages in the functional areas.
- Identify the benefits that the proposed software system brings to organization as a whole.
- Align the stated advantages to corporate goals of the organization.
- Use the corporate language in stating the advantages.
- If possible, do a payback analysis or break-even analysis to show how soon the organization could expect to recover the cost of the system. An amortization schedule would be useful here.
- Briefly mention risks or drawbacks (these will be discussed in the feasibility analysis).

3.6 Feasibility Analysis Report

A feasibility study is a thorough analysis of a prospective project to determine the likelihood of success of the venture. Traditionally, the analysis is done in three areas: *the technical feasibility, the economic feasibility and the operational feasibility*. Additionally, the alternate solutions to the defined problem must be

carefully examined, and recommendation made in favor of the most prudent alternative.

3.6.1 Technical Feasibility

The technical feasibility analysis addresses the basic question of whether the project is technically feasible. The analysis must be done with respect to four main considerations. Once these four areas are addressed, the question of technical feasibility can be answered with reasonable accuracy. The areas are as follows:

- Availability of hardware required
- Availability of software required
- Availability of knowledge and expertise
- Availability of other technology that may be required e.g. telecommunication, new software development tool, etc.

3.6.2 Economic Feasibility

The economic feasibility analysis is aimed at measuring the economic results that would materialize if the project is pursued. Important considerations are

- Costs versus benefits expressed in terms of payback period, cash flow and return on investment (ROI)
- Development time and the financial implications
- Economic life — for how long the system will be economically advantageous
- Risk analysis — examining the associated risks

The matter of risk is very important, and needs some elucidation. Every business venture has associated with it, some inherent risks; software engineering projects are not excluded from this reality. The software engineer should be aware of the following:

- Generally speaking (from a financial perspective), the higher

the risks are, the greater is the possible return on investment (and vice versa). However, there are exceptions.

- In many instances, taking prudent precautionary steps can reduce software engineering project risks. The software engineer must evaluate each risk and determine whether it can be reduced.

Two commonly used business techniques for evaluating and analyzing risks are *payback analysis* and return on investment (ROI). ROI is often facilitated by what is called a *net present value* (NPV) analysis. [Figure 3-1](#) provides the formulas used in payback analysis and ROI respectively.

1. $PB = \text{Investment} / [\text{Average Cash flow per Annum}]$
2. $NPV = -A_0 + A_1 / (1+r_1) + A_2 / (1+r_2)^2 + A_n / (1+r_n)^n$
Key
PB = Pay-back
NPV = Net Present-value
A_0 = Initial investment
A_i = Annual cash flow in the form of savings and/or additional income
r_i = Rate of return (may be constant)

[Figure 3-1](#). Formulae for Payback and ROI

3.6.3 Operational Feasibility

The operational feasibility speaks to issues such as the likely user/market response to the system, as well as the required changes on the part of operational staff. Generally speaking, user involvement in a project tends to positively impact user acceptance. The factors to be considered in assessing the operational feasibility of a project include the following:

- User attitude to (likely) changes
- Likelihood of changes in organizational policies
- Changes in methods and operations that may be necessary
- Timetable for implementation (long implementation time implies low feasibility)

3.6.4 Evaluation of System Alternatives

Evaluation of system alternatives should be based on software quality factors and feasibility factors. In the previous sub-section, we looked at the feasibility factors. Let us now examine the quality factors, explore the cost factors a bit further, and see how the evaluation is done.

Evaluation Based on Quality

The quality factors of [chapter 1](#) are relevant here. In the interest of clarity, they are repeated:

- **Maintainability:** How easily maintained is the software? This will depend on the quality of the design as well as the documentation.
- **Documentation:** How well documented is the system?
- **Efficiency:** How efficiently are certain core operations carried out? Of importance are the response time and the system throughput.
- **User Friendliness:** How well designed is the user interface? How easy is it to learn and use the system?
- **Compatibility** with existing software products.
- **Security:** Are there mechanisms to protect and promote confidentiality and proper access control?
- **Integrity:** What is the quality of data validation methods?
- **Reliability:** Will the software perform according to requirements at all times?
- **Growth potential:** What is the storage capacity? What is the capacity for growth in data storage?
- **Functionality and Flexibility:** Does the software provide the essential functional features required for the problem domain? Are there alternate ways of doing things?
- **Adaptability:** How well are unanticipated situations

handled?

- **Differentiation:** What is unique about the software?
- **Productivity:** How will productivity be affected by the software?
- **Comprehensive Coverage:** Is the problem comprehensively and appropriately addressed?

Since these factors are qualitative, weights usually are applied, and each alternative is assessed on each factor. The total grade for each alternative is then noted. The result may be neatly presented in tabular form as in [Figure 3-2](#). In this illustration, alternative-B would be the best option.

Quality Factors	System Alternatives		
	Alternative-A	Alternative-B	Alternative-C
Maintainability [Max 10]	8	9	9
Efficiency [Max 10]	9	9	6
User Friendliness [Max 10]	7	9	6
Documentation [Max 10]	5	8	10
Compatibility [Max 10]	4	8	9
Security [Max 10]	4	9	9
Reliability [Max 10]	5	8	9
Flexibility & Functionality [Max 10]	8	9	9
Adaptability [Max 10]	8	9	10
Growth Potential [Max 10]	6	9	7
Productivity [Max 10]	6	9	7
Overall Evaluation Score	70	96	85

Recommended alternative is Alternative-B.

[Figure 3-2](#). Quality Evaluation Grid Showing Comparison of System Alternatives

Evaluation Based on Cost

The main cost components to be considered are development cost, operational cost, equipment cost, and facilities cost.

Engineering costs include

- Investigation, Analysis and Design cost
- Development cost

- Implementation cost
- Training cost

Operational costs include

- Cost of staffing
- Cost for data capture and preparation
- Supplies and stationery cost
- Maintenance cost

Equipment costs include

- Cost of hardware
- Cost of software
- Cost of transporting equipment
- Depreciation of equipment over the acquisition and usage period

3.6.5 Evaluation of System Alternatives (continued)

Facilities costs are sometimes bundled with operational cost. If there is a distinction, then facilities costs would typically include

- Computer installation cost
- Electrical and cooling cost
- Security Cost

A similar analysis may be constructed for the feasibility factors (including cost factors). When the alternatives are evaluated against these feasibility factors, an evaluation matrix similar to that shown in [Figure 3-3](#) may be constructed. In this illustration, alternative-C would be the best option. Of course, the two Figures ([Figures 3-2](#) and [3-3](#)) may be merged.

Feasibility Factors		Alt-A	Alt-B	Alt-C
Technical Feasibility		68	60	78
Availability of Hardware (bigger means better)	[Max 20]	20	15	20
Availability of Software (bigger means better)	[Max 20]	18	15	20
Availability of Expertise (bigger means better)	[Max 20]	14	16	20
Availability of Technology (bigger means better)	[Max 20]	16	12	18
Economic Feasibility		106	107	108
Engineering Cost (bigger means lower cost)	[Max 20]	18	17	17
Equipment Cost (bigger means lower cost)	[Max 20]	17	17	17
Operational Cost (bigger means lower cost)	[Max 20]	19	19	19
Facilities Cost (bigger means lower cost)	[Max 20]	19	19	19
Development Time (bigger means shorter time)	[Max 20]	18	18	19
Economic Life (bigger means longer time)	[Max 20]	16	18	18
Risk (bigger means lower risk)	[Max 20]	18	18	18
Operational Feasibility		50	52	56
User Attitude to Likely Changes (bigger means better)	[Max 20]	20	16	18
Likelihood of Organizational Policy Changes (bigger means fewer changes)	[Max 20]	16	19	18
Implementation Time (bigger means shorter time)	[Max 20]	14	17	20
Overall Evaluation Score	[Max 280]	243	238	261

Figure 3-3. Feasibility Evaluation Grid Showing Comparison of System Alternatives

Putting the Pieces Together

A final decision is taken as to the most prudent alternative, after examination of each alternative with respect to the quality factors and the feasibility factors. The rule of thumb is to recommend the alternative with the highest score. However, other constraints such as financial constraints, focus of the organization, etc., may mitigate against that option. Given the overall organizational situation, the most prudent and realistic alternative is taken.

3.7 Alternate Approach to the Feasibility Analysis

A popular alternate approach to conducting the feasibility analysis involves the consideration of the feasibility factors, the quality factors and the cost factors in a rather comprehensive way. The evaluation criteria are grouped in the following categories:

- TELOS Feasibility Factors: Technical, Economic, Legal, Operational, Schedule feasibility factors

- PDM Strategic factors: Productivity, Differentiation and Management factors
- MURRE Design Factors: Maintainability, Usability, Reusability, Reliability, Extendibility factors

The principle of weights and grades described in the previous section is also applicable here. [Figure 3-4](#) illustrates a sample rating worksheet that could be employed.

TELOS Feasibility Factors:					
	A	B	C	D	
Technical Feasibility					
Economic Feasibility					
Legal Feasibility					
Operational Feasibility					
Schedule Feasibility					
Subtotal					

PDM Feasibility Factors:					
	A	B	C	D	
Productivity					
Differentiation					
Management					
Subtotal					

MURRE Feasibility Factors:					
	A	B	C	D	
Maintainability					
Usability					
Reusability					
Reliability					
Extendibility					
Subtotal					
GRAND TOTAL					

[Figure 3-4](#). Alternate Evaluation Matrix for Project Alternatives

3.8 Summary of System Inputs and Outputs

It is sometimes useful to provide a summary of the main inputs to a software

system. Depending on the category and complexity of the software, the inputs may vary from a small set of electronic files to a large set involving various input media. Input and output design will be discussed in more detail in [chapter 11](#). For now, and at this point in the project, a simple list of system inputs will suffice.

Additionally, a brief indication of what to expect from the system from a user viewpoint is sometimes useful in giving the potential end-user a chance to start identifying with the product even before its development has begun. Usually, a summary of possible system outputs will suffice.

In situations where visual aid is required, a non-operational prototype (see [chapter 5](#)) might be useful. For a real-time system (see [chapter 13](#)), a simulation may be required.

3.9 Initial Project Schedule

A preliminary project schedule outlining the estimated duration of project activities (from that point) is usually included in the ISR. Please note:

- The initial schedule may be detailed or highly summarized depending on information available at the time of preparation. As such, subsequent revision may be necessary (during the design specification).
- A PERT diagram, Gantt chart (see [chapter 8](#)), or simple table will suffice.
- A cost schedule is also provided showing projected costs for various activities. This may also be subject to subsequent revision.

[Figure 3-5](#) illustrates a sample tabular representation of a project schedule. As illustrated in the figure, it is a good idea to include in the schedule, an estimate of the number of professionals who will be working on the project. As more details become available, it will be necessary to revise this schedule; but for now, a tabular representation (as shown in the figure) will suffice. [Chapter 8](#) discusses project scheduling in more detail.

Activity	Activity Description	Estimated Time (Days)	Estimated Cost (\$)
A	Design System Architecture	30	72,000
B	Design Operation Specifications	12	28,800
C	Design Control Operations	8	19,200
D	Design Modification Operations	15	36,000
E	Design Inquiry/Report Operations	7	16,800
F	Code Control Operations	2	4,800
G	Prepare System Users Guide	5	12,000
H	Test Control Operations	2	4,800
I	Code Modification Operations	6	14,400
J	Test Modification Operations	4	9,600
K	Code Inquiry Operations	2	4,800
L	Code Report Operations	3	7,200
M	Test Inquiry/Report Operations	1	2,400
N	Integration Test	4	9,600
Total estimated time and cost		91	242,400
Estimated Number of Fulltime Professionals is 3			

Figure 3-5. Sample Initial Project Schedule

3.10 Project Team

The final component of your ISR is a proposed project team. The team must be well balanced; typically it should include the following job titles:

- Project Manager
- Secretary
- Systems Analyst(s) or Software Engineer(s)
- Programmer(s) (if different from software engineers)
- Software Documenter(s) (if different from software engineers)
- Data Clerk(s)
- User Interface Officer(s)

The number of personnel and the chosen organization chart will depend on the size and complexity of the project. Additionally, a user work group or steering committee is normally set up to ensure that the stated objectives are met. This may include the following personnel:

- Director of Information Technology (if different from Project Manager)
- Project Manager
- Representation from the principal users
- Other managers whose expertise may be required from time to time (on invitation)
- Software Engineer(s) (on invitation)
- Managing Director (depending on nature of the project)

In some instances, the project team forms a sub-committee of the steering committee; in small organizations, the two teams may be merged. [Figure 3-6](#) shows three possible traditional configurations of the project team. As an exercise, you are encouraged to identify the pros and cons of each project team configuration. It must be pointed out however, that there may be various other configurations of a software engineering team ([Chapter 19](#) provides more clarification).

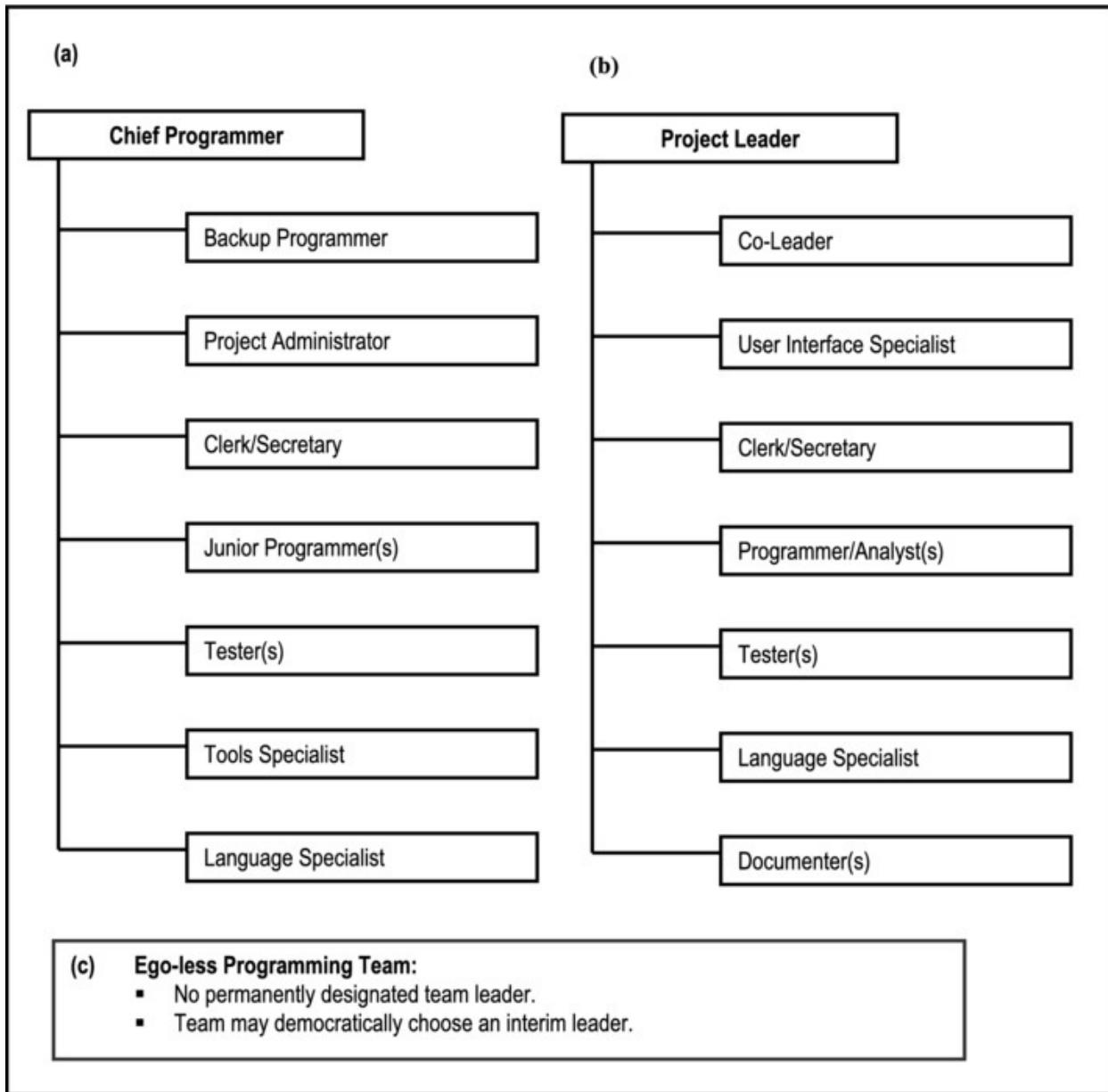


Figure 3-6. Possible Arrangements of the Project Team

3.11 Summary and Concluding Remarks

It's now time to summarize what we have covered in this chapter:

- Selection of a software engineering project depends on the

corporate mission and objectives of the organization.

- The ISR is the first deliverable of the software engineering project. It consists of a number components such as problem definition, proposed solution, scope and objectives, system justification, feasibility analysis report, summary of system outputs, initial project schedule, and the project team.
- Problem definition sets the tone for the project. It must therefore be done thoroughly and accurately.
- The proposed solution provides a summary of how the problem will be addressed.
- The system scope defines the extent and boundaries of the software system; system objectives are also clearly specified.
- The system justification provides the rationale and main benefits of the software system.
- The feasibility analysis report analyzes and reports on the likelihood of success of the project. The feasibility study examines the technical, economic and operational feasibility, as well as other related factors such as associated risks, maintainability, usability, reusability, reliability, extendibility, productivity, differentiation, management issues, legal issues, functionality, flexibility, security, etc.
- Summary of system inputs and outputs is self-explanatory.
- The initial project schedule provides a draft schedule of the project activities that will subsequently be updated as more detailed information becomes available.
- The project team indicates the individuals that are or will be involved in the project.

[Appendix 8](#) provides excerpts from the ISR for an inventory management system. Please take some time to carefully review the document and gain useful insights.

Once the ISR is completed, this document is submitted to the management of the organization for consideration and approval. Typically, the software engineer responsible for its preparation may be required to make a formal presentation to the organization. In this presentation, you sell the project idea to the

organization. Once approval is obtained, you then start working on your next deliverable which is the *requirements specification* (RS). The next few chapters will prepare you for this. As you proceed, bear in mind that one primary objective of software engineering is to produce software that has a significantly greater value than its cost. We do this by building value in the areas of the software's quality factors.

3.12 Review Questions

1. What factors are likely to influence the selection of a software engineering project? Briefly discuss these factors, and explain why they are important.
2. Discuss the importance of problem definition in a software engineering project.
3. What should the proposed solution entail?
4. What information is provided when you state system scope and system objectives?
5. How do you justify the acquisition of a new software system in an organization?
6. What is a feasibility study? Discuss the main components of a feasibility report.
7. Describe an approach that is commonly used for evaluating system alternatives and determining the most prudent alternative.
8. Identify three approaches to organizing a project team. For each approach, discuss its strengths and weaknesses, and describe a scenario that would warrant such approach.
9. Conduct an examination of an organization that you are familiar with, and do the following:
 - Propose an organization chart for the organization.
 - By examining the organization chart, propose a set of software systems that you would anticipate for this organization.

- Choose one of those potential systems, and for this system, develop an ISR as your first deliverable. Your ISR must include all the essential ingredients discussed in the chapter.

3.13 References and/or Recommended Readings

[Burch, 1992] Burch, John G. *Systems Analysis, Design and Implementation*. Boston, MA: Boyd & Fraser, 1992.

[Harris, 1995] Harris, David. *Systems Analysis and Design: A Project Approach*. Fort Worth, TX: Dryden Press, 1995. See [chapter 3](#).

[Kendall, 2005] Kendall, Kenneth E., and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [chapter 3](#).

[Long, 1989] Long, Larry. *Management Information Systems*. Eaglewood Cliffs, NJ: Prentice Hall, 1989. See [chapters 10-12, 17](#).

[Peters, 2000] Peters, James F. and Witold Pedrycz. *Software Engineering: An Engineering Approach*. New York, NY: John Wiley & Sons, 2000. See [chapter 4](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapter 3](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [chapter 4](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See [chapter 4](#).

[Van Vliet, 2000] Van Vliet, Hans. *Software Engineering* 2nd ed. New York, NY: John Wiley & Sons, 2000. See [chapter 2](#).

CHAPTER 4



The Requirements Specification

This chapter introduces the second major deliverable in a software engineering project — the requirements specification. It provides an overview of the deliverable, and prepares you for details that will be covered in the next four chapters. The chapter proceeds under the following subheadings:

- Introduction
- Contents of the Requirements Specification
- Documenting the Requirements
- Requirements Validation
- How to Proceed
- Presenting the Requirements Specification
- Summary and Concluding Remarks

4.1 Introduction

The software *requirements specification* (RS) is the second major deliverable in a software engineering project. It is a document that signals the end of the investigation and analysis phase, and the beginning of the design phase. Its contents will serve as a blueprint during design and development.

 **Note** Remember, we are not assuming that life cycle model being employed is necessarily the waterfall model, which is irreversible. Rather, please assume that any of the reversible life cycle models is also applicable.

The requirements specification must be accurate as well as comprehensive, as it affects the rest of the system's life in a significant way. Any flaw in the requirements specification will put pressure on the subsequent phases (particularly design and development) to attempt to address it. If the problem is not identified and addressed by the end of system development, then it must be addressed during the maintenance phase — a particularly undesirable situation. In some instances, this might be too late to attempt saving a reputation of a software product.

The requirements specification serves as a very effective method of system documentation, long after system development.

Note Many texts on software engineering recommend one comprehensive requirements specification document that involves detailed design issues. From experience, this is not a good idea, especially if the software being developed is very large and complex. The approach recommended by this course is to have a requirements specification, which provides comprehensive and accurate coverage of the system, but avoids fine design intricacies (such as characteristics of data elements, operation specifications, use of technical language for business rules, system names for objects, etc). This blueprint is then used as input to the design phase, where all intricacies are treated in a separate *design specification*.

As you go through this chapter, bear in mind that you will need to gain mastery of a number of techniques in order to develop a requirements specification for a project. The intent here is to provide you with the big picture, so you will have an appreciation of where you are heading. [Chapters 5 through 8](#) will then work you through the required details.

4.2 Contents of the Requirements Specification

The requirements specification is comprised of a number of important ingredients. These will be briefly described in this section.

Acknowledgments: This is recognition of individuals and/or organizations

or offices that significantly contributed to the achievement of the deliverable. Typically, it is written last but appears as the first item in the requirements specification.

Problem Synopsis: This involves the problem statement, proposed system solution, and system objectives: having been drawn from the ISR (see [chapter 3](#)), these would be refined by now. The problem synopsis should also involve a brief review of feasibility findings (the details you can include as appendix).

System Overview: This should provide a broad perspective of the software system. An *information topology chart* (ITC) and/or an *object flow diagram* (OFD) are/is particularly useful here. These techniques will be discussed in [chapter 6](#).

System Components: Since your prime objective is to provide clarity about the requirements of the software being engineered, it is a good habit to provide a list of the subsystems, modules, entities (object types) and operations.

Detailed Requirements: This involves specification of the software requirements in a concise, unambiguous manner. This involves the use of narrative, flow-charts, diagrams and other techniques where applicable. These techniques will be discussed in [chapters 6 and 7](#). Depending on the size and complexity of the project, the requirements may be specified for the system as a whole (if the system is small), for subsystems comprising the system (in the case of medium sized to large systems), or for modules comprising subsystems (if the subsystems are large). An approach that has been successfully employed by the author is to include the following in the detailed requirements:

- **Storage Requirements:** The information entities (object types) and how they are comprised.
- **Operational Requirements:** The operations to be defined on the information entities (object types).
- **Business Rules:** The rules of operation. These include derivation rules, relationship integrity rules, data integrity rules, control conditions, and trigger rules. Rules are more thoroughly discussed [chapter 7](#).

Interface Specification: This involves the guidelines relating to how various subsystems and/or modules will be interconnected. Depending on the project, this may or may not be critical at this point. In many cases, it can be deferred until the design phase (in fact, it will be elucidated later in the course).

System Constraints: This involves the operating constraints of the system.

Like the interface specification, this requirement may be trivial and self-evident, or critical to the success of the software. An example of triviality would be in the case of an information system (for instance, an inventory management system) for an organization. An example of situation where system constraints would be critical is a real time system (for instance managing atmospheric pressure in an air craft, or water pressure for a water power station).

Revised Project Schedule: Here, you provide a refined project schedule, which involves a PERT diagram or Gantt chart, or some other equivalent means. This will be discussed in [chapter 8](#).

System Security Requirements: Depending on the intended purpose and use of the software product, system security will be an issue to be considered. In specifying the security requirements, the following must be made clear:

- What users will have access to the system
- What system resources various users can access and how
- What restrictions will apply to various users

Concluding Remarks: Usually, there are some concluding remarks about the requirements specification. These are placed here.

Appendices: The appendices typically include supporting information that are relevant to the software requirements, but for various reasons might have been omitted from the body of the document. Examples of possible information for the appendices are

- Investigation and analysis team
- Proposed design and development team(s)
- Photocopy of approval(s) from end-user and management
- Photocopies of source documents used in investigation and analysis
- Detailed feasibility report

4.3 Documenting the Requirements

Documenting the requirements will depend to a large extent on the intended

readership and the software tools available. If the intended readership is strictly technical people such as software engineers, then unnecessary details can be avoided, technical jargons are allowed and formal methods may be used (more on this later). If the intended readership includes non-technical individuals (lay persons), then it is advisable to develop an easy to read document.

The requirements specification may be developed with inexpensive applications such as MS Office, Corel Suite, etc. If a CASE tool is available, it should be used since this will enhance the system development process. In fact, in many cases, the diagrams used in the CASE tool are executable — code is generated from them — so that the distinction of requirements specification from design and design from development are absent (hence the term modeling). However, in many instances, thorough documentation is achieved by employing a mix of software development tools and desktop applications.

Preparation of the requirements specifications is perfected only as the software engineer practices on the job. Whether a CASE tool or desktop publisher is used, you will need to have mastery of fundamental software engineering principles, if the end product is to be credible.

4.4 Requirements Validation

Once the requirements have been specified, they must be validated. This is crucial, since it is these requirements that will be used as the frame of reference for the next deliverable - the design specification. If the requirement specification is lacking in content or accuracy, this will have adverse repercussions on the design and final product. There are two approaches to requirements validation: manual and automatic.

Manual Techniques: In manual review, a resource team is used to check the following:

- Each specification must be traceable to a requirement definition
- Each requirement definition is traceable to a specification
- All required specifications are in place
- All system objectives are met

The techniques available for this (these will be discussed in the next chapter)

are

- Interviews
- Questionnaires
- Document review
- Sampling and Experimenting
- Observation
- Prototyping
- Brainstorming
- Manual cross-referencing
- Mathematical proofs

Automatic Techniques: In automatic techniques, an attempt is made to automate the validation process. This involves technique such as cross-referencing, prototyping, automated simulations models, and, mathematical proofs. Automatic validation is not always feasible, but when applied, is quite helpful.

Discuss: Which life cycle model would most favor automatic validation techniques?

4.5 How to Proceed

How you proceed to develop the RS is a matter of personal preference. Some people like a top-down approach; others prefer a bottom-up approach. Traditional systems employed functional-oriented methods; contemporary systems tend to favor object-oriented methods. In this course, you will be exposed to both approaches, even though there is a bias to the object-oriented approach.

It preparing the RS, it is recommended that you be methodical and incremental, revisiting formerly addressed issues whenever you need to. [Figure 4-1](#) describes a recommended approach that you will find useful in various scenarios. You will notice a few newly introduced terms; these will be clarified in the upcoming chapters.

1. Refine your problem definition and proposed software solution
2. Conduct your information gathering and identify:
 - Subsystem(s) and/or modules
 - Object types or *information entities* (clarified in chapters 5 and 10)
3. Identify or define operations for the various object types.
4. Provide a system overview which includes any convenient combination of the following:
 - System narrative
 - *Information topology chart* (clarified in chapter 6)
 - *Object flow diagram* (clarified in chapter 6)
5. For each subsystem, provide the following:
 - System overview (as in item 4)
 - *Entity-Relationship* (E-R) or *Object-Relationship* (O-R) diagram (clarified in chapter 10)
 - Storage requirements
 - Operational requirements
 - System rules
 - Interface requirements (if applicable)
 - System constraints (if applicable)

The approach for real time systems would be slightly different. This will be discussed later in the course.

Figure 4-1. How to Prepare the Requirements Specification

4.6 Presentation of the Requirements Specification

It is often required that the requirements specification be presented to a group of experts and possible end users for scrutiny. Should you have the responsibility to do this, below are some guidelines:

- Have a summarized version for the purpose of presentation.
- Make the presentation interesting by using a balanced mix of the various resources at your disposal.
- Have photocopies for the critical audience.
- Make allocation and be prepared for questions.
- Here are some positive tips:
 - Be thoroughly prepared
 - Project loudly

- Have a positive eye contact
- Make visuals large enough for audience to see
- Speak in your natural language, tone style
- Be confident and calm

Your presentation must be aesthetically attractive and compelling. As such, you should make use of an appropriate presentation software product (examples include Microsoft PowerPoint, Microsoft Publisher, Corel Draw, Page Maker, etc.).

4.7 Summary and Concluding Remarks

Let us summarize what we have covered in this chapter:

- The RS is the second major deliverable of a software engineering project. It must be accurate and comprehensive.
- The RS typically consists of the following components: acknowledgments, problem synopsis, system overview, detailed requirements, interface specification, system constraints, revised project schedule, security requirements, conclusion, and appendices.
- The requirements may be documented using a CASE tool or a desktop processing application.
- Requirements validation ensures that the software requirements are accurate. The process may be manual or automatic.
- In preparing the RS, it is recommended that you be methodical and incremental in your approach.
- You may be required to make a formal presentation of the RS. Preparation is of paramount importance.

[Appendix 9](#) provides excerpts from the RS of the inventory management

system that was mentioned in the previous chapter. Take some time to review it and gain useful insights. But to get to that point where you are able to prepare such a document on your own, you need to learn some more fundamentals, commencing with information gathering techniques. This will be discussed in the next chapter.

4.8 Review Questions

1. Explain the importance of the requirements specification.
2. Identify six important ingredients of the requirements specification. For each, briefly explain what it is and why it is important.
3. How important is requirements validation? Describe two approaches to requirements validation.
4. Outline clearly, how you would proceed to develop a requirements specification for a software engineering project.

4.9 References and/or Recommended Readings

[Kendall, 1995] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design 3rd ed.* Upper Saddle River, NJ: Prentice Hall, 1995. See [chapters 13](#) and [14](#).

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development With UML and Java.* Upper Saddle River, NJ: Prentice Hall, 2002.

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design.* Eaglewood Cliffs, NJ: Pretence Hall, 1993.

[Peters, 2000] Peters, James F. and Witold Pedrycz. *Software Engineering: An*

Engineering Approach. New York, NY: John Wiley & Sons, 2000. See [chapter 3](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, 2006. [Chapter 4](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [chapter 10](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See [chapter 6](#).

CHAPTER 5



Information Gathering

In order to accurately and comprehensively specify the system, the software engineer gathers and analyzes information via various methodologies. This chapter discusses these methodologies as outlined below:

- Rationale for Information Gathering
- Interviews
- Questionnaires and Surveys
- Sampling and Experimenting
- Observation and Document Review
- Prototyping
- Brainstorming and Mathematical Proof
- Object Identification
- Summary and Concluding Remarks

5.1 Rationale for Information Gathering

What kind of information is the software engineer looking for? The answer is simple, but profound: You are looking for information that will help to accurately and comprehensively define the requirements of the software to be constructed. The process is referred to as *requirements analysis*, and involves a

range of activities that eventually lead to the deliverable we call the *requirements specification* (RS). In particular, the software engineer must determine the following:

- **Synergistic interrelationships** of the system components: This relates to the components and how they (should) fit together.
- **System information entities** (*object types*) and their interrelatedness: An entity refers to an object or concept about data is to be stored and managed. Entities and object types will be further discussed in [chapters 9 and 10](#).
- **System operations** and their interrelatedness: Operations are programmed instructions that enable the requirements of the system to be met. Some operations are system-based and may be oblivious to the end user; others facilitate user interaction with the software; others are internal and often operate in a manner that is transparent to the end user.
- **System business rules:** Business rules are guidelines that specify how the system should operate. These relate to data access, data flow, relationships among entities, and the behavior of system operations.
- **System security mechanism(s)** that must be in place: It will be necessary to allow authorized users to access the system while denying access to unauthorized users. Additionally, the privileges of authorized users may be further constrained to ensure that they have access only to resources that they need. These measures protect the integrity and reliability of the system.

As the software engineer embarks on the path towards preparation of the requirements specification, these objectives must be constantly borne in mind. In the early stages of the research, the following questions should yield useful pointers:

- WHAT are the (major) categories of information handled? Further probing will be necessary, but you should continue your pursuit until this question is satisfactorily answered.

- WHERE does this information come from? Does it come from an internal department or from an external organization?
- WHERE does this information go after leaving this office? Does it go to an internal department or to an external organization?
- HOW and in WHAT way is this information used? Obtaining answers to these questions will help you to identify business rules (to be discussed in [chapter 7](#)) and operations (to be discussed in [chapters 11 and 12](#)).
- WHAT are the main activities of this unit? A unit may be a division or department or section of the organization. Obtaining the answer to this question will also help you to gain further insights into the operations (to be discussed in [chapters 11 and 12](#)).
- WHAT information is needed to carry out this activity? Again here, you are trying to refine the requirements of each operation by identifying its input(s).
- WHAT does this activity involve? Obtaining the answer to this question will help you to further refine the operation in question.
- WHEN is it normally done? Obtaining the answer to this question will help you to further refine the operation in question by determining whether there is a time constraint on an operation.
- WHY is this important? WHY is this done? WHY...? Obtaining answers to these probes will help you to gain a better understanding of the requirements of the software system.

Of course, your approach to obtaining answers to these probing questions will be influenced by whether the software system being researched is to be used for in-house purposes, or marketed to the public. The next few sections will examine the commonly used information gathering strategies.

5.2 Interviewing

Interviewing is the most frequent method of information gathering. It can be very effective if carefully planned and well conducted. It is useful when the information needed must be elaborate, or clarification on various issues is required. The interview also provides an opportunity for the software engineer to win the confidence and trust of clients. It should therefore not be squandered.

Steps in Planning the Interview

In planning to conduct an interview, please observe the following steps:

1. Read background information.
2. Establish objectives.
3. Decide whom to interview.
4. Prepare the interviewee(s).
5. Decide on structure and questions.

Basic Guidelines for Interviews

Figure 5-1 provides some guidelines for successfully planning and conducting an interview. These guidelines are listed in the form of a do-list, and a don't-list.

Do-List for Interviews:

1. Make an appointment.
2. Plan. Consider topics to be covered.
3. Make sure information requested is impersonal and objective.
4. Prepare for the interview (theme & questions).
5. Ask questions at the right level.
6. State the purpose clearly, up front.
7. Communicate in the interviewee's language.
8. If compliments become necessary, be sincere.
9. Be relaxed and help the respondent to be relaxed.
10. Listen.
11. Identify facts as opposed to opinions. Both are important.
12. Accept ideas and hints.
13. Check the facts.
14. Collect source documents and forms.
15. Make effective use of open-ended and closed questions.
16. Part pleasantly.

Don't-List for Interviews:

1. Don't be late.
2. " be too formal or too casual.
3. " interrupt the speaker.
4. " use technical jargons.
5. " jump to conclusions.
6. " argue or criticize (constructive or destructive).
7. " make suggestions (not as yet; you will get your opportunity to do so later).

Figure 5-1. Basic Guidelines for Interview

5.3 Questionnaires and Surveys

A questionnaire is applicable when any of the following situations hold:

- A small amount of information is required of a large population.
- The time frame is short but a vast area (and/or dispersed population) must be covered.
- Simple answers are required to a number of standard questions.

5.3.1. Guidelines for Questionnaires

Figure 5-2 provides a set of basic guidelines for preparing a questionnaire.

- | | |
|--|---|
| 1. State purpose clearly
2. Thank the participants
3. Must have a topic or heading which reflects an apt summary of the information sought.
4. Should adhere to the principles of forms design.
5. Avoid ambiguity.
6. Decide when to use open-ended questions, closed questions or scalar questions.
7. Order questions appropriately.
8. State questions in a language the respondent will readily understand.
9. Be consistent in style.
10. Ask questions of importance to the respondents first.
11. Bring up less controversial questions first.
12. Cluster related questions. | } Usually in the form of a cover note or letter |
|--|---|

Figure 5-2. Guidelines for Questionnaires

5.3.2. Using Scales in Questionnaires

Scales may be used to measure the attitudes and characteristics of respondents, or have respondents judge the subject matter in question. There are four forms of measurement as outlined below:

1. **Nominal Scale:** Used to classify things. A number represents a choice. One can obtain a total for each classification.
2. **Ordinal Scale:** Similar to nominal, but here the number implies ordering or ranking.

Example 1: The financial status of an individual may be represented as follows:

1 = Extremely Rich; 2 = Very Rich; 3 = Rich; 4 = Not Rich; 5 = poor; 6 = very poor; 7 = Pauper

3. **Interval Scale:** Ordinal with equal intervals.

Example 2: Usage of a particular software product by number of modules used (10 means high):

1 2 3 10

4. **Ratio Scale:** Interval scale with absolute zero.

Example 3: Distance traveled to obtain a system report:

0 1 2 3 10 [Meters]

Example 4: Average response time of the system:

0 1 2 3 10 [Minutes]

5.3.3. Administering the Questionnaire

Options for administering the questionnaire include the following:

- Convening respondents together at one time
- Personally handing out blank questionnaires and collecting completed ones

- Allowing respondents to self-administer the questionnaire at work and leave it at a centrally located place
- Mailing questionnaires with instructions, deadlines, and return postage
- Using the facilities of the World Wide Web (WWW), for example e-mail, user forums and chat rooms

5.4 Sampling and Experimenting

Sampling is useful when the information required is of a quantitative nature or can be quantified, no precise detail is available, and it is not likely that such details will be obtained via other methods. [Figure 5-3](#) provides an example of a situation in which sampling is relevant.

Sampling theory describes two broad categories of samples:

- *Probability sampling* involving random selection of elements
- *Non-probability sampling* where judgment is applied in selection of elements

Shipping	Number of Orders	% of Total
As Promised	186	37.2
1 day late	71	14.2
2 days late	49	9.8
3 days late	35	7.0
4 days late	38	7.6
5 days late	28	5.6
6 days late	93	18.6
	500	100.0

[Figure 5-3](#). Examining the Delivery of Orders after Customer Complaints

5.4.1 Probability Sampling Techniques

There are four types of probability sampling techniques:

- **Simple Random Sampling** uses a random method of selection of elements.
- **Systematic Random Sampling** involves selection of elements at constant intervals. Interval = N/n where N is the population size and n is the sample size.
- **Stratified Sampling** involves grouping of the data in strata. Random sampling is employed within each stratum.
- **Cluster Sampling:** The population is divided into (geographic) clusters. A random sample is taken from each cluster.

Note The latter three techniques constitute *quasi-random sampling*. The reason for this is that they are not regarded as perfectly random sampling.

5.4.2 Non-probability Sampling Techniques

There are four types of non-probability sampling techniques:

- **Convenience Sampling:** Items are selected in the most convenient manner available.
- **Judgment Sampling:** An experienced individual selects a sample (e.g. a market research).
- **Quota Sampling:** A subgroup is selected until a limit is reached (e.g. every other employee up to 500).
- **Snowball Sampling:** An initial set of respondents is selected. They in turn select other respondents; this continues until an acceptable sample size is reached.

5.4.3 Sample Calculations

[Figure 5-4](#) provides a summary of the formulas that are often used in performing calculations about samples:

Item	Clarification
Mean	$X' = \sum(X)/n$ OR $\sum(F_i X_i) / \sum(F_i)$ where n is the number of items (elements); F_i is the frequency of X_i ; and X_i represents the data values.
Standard Deviation	$S = \sqrt{(\sum F_i (X_i - X')^2) / n}$ where n is the sample size
Variance	$\text{Variance} = S^2$
Standard Unit	$Z = (X_i - X') / S$
Standard Error	$S_E = S / \sqrt{n}$
Unit Error	$r = ZS_E = ZS / \sqrt{n}$
Sample Size	From the equation for unit error above, $n = (ZS / r)^2$

Figure 5-4. Formulas for Sample Calculations

These formulas are best explained by examining the normal distribution curve ([Figure 5-5](#)). From the curve, observe that:

- Prob (-1 \leq Z \leq 1) = 68.27%
- Prob (-2 \leq Z \leq 2) = 95.25%
- Prob (-3 \leq Z \leq 3) = 99.73%

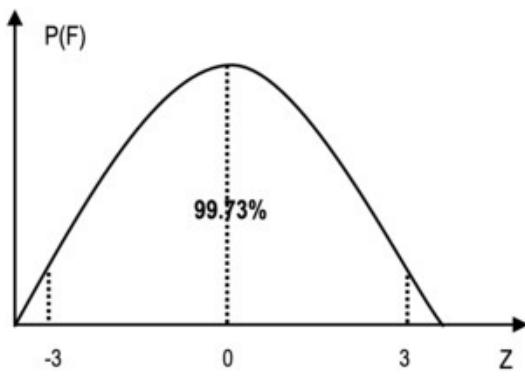
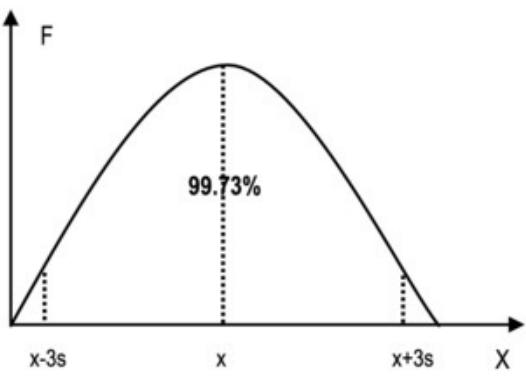
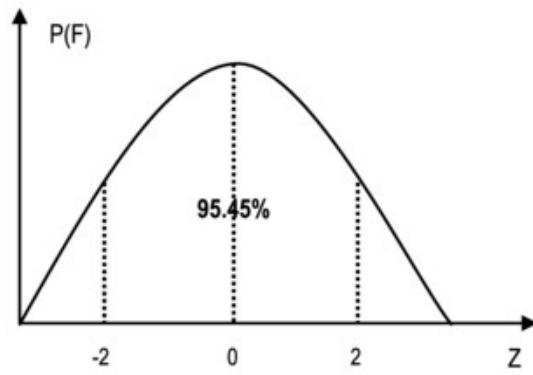
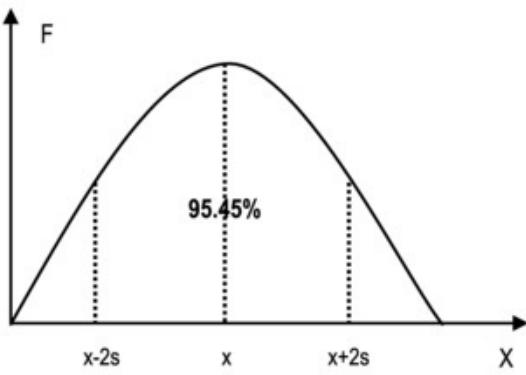
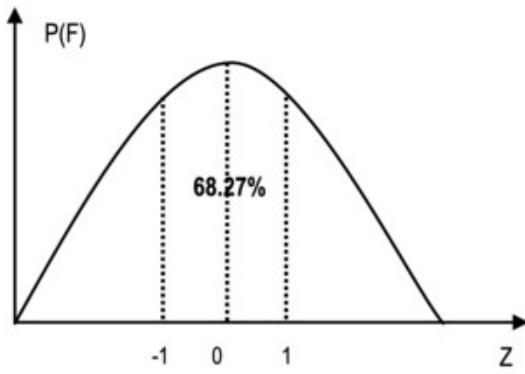
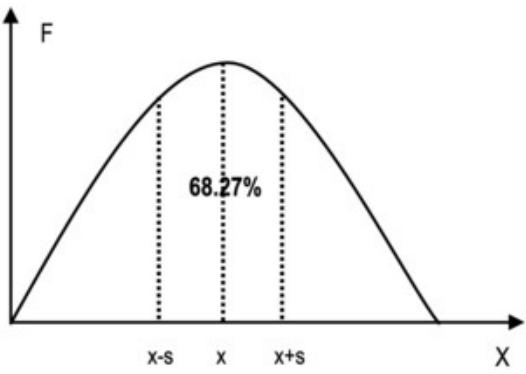
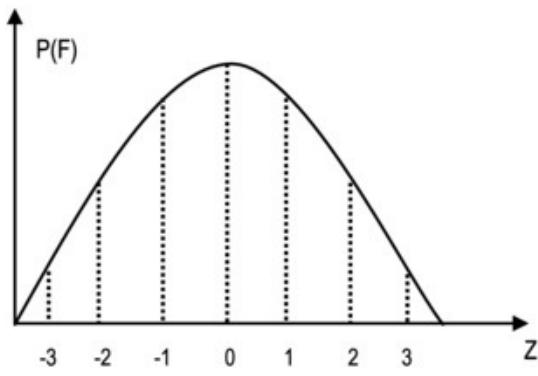
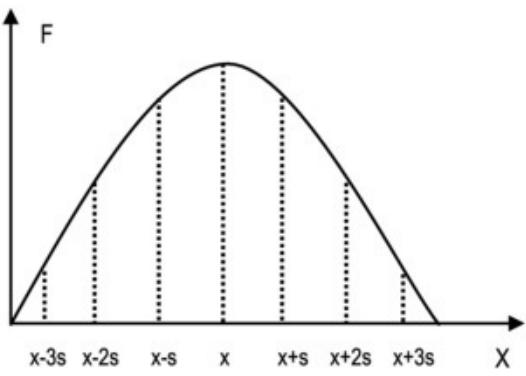


Figure 5-5. The Normal Distribution Table

The confidence limit of a population mean is normally given by $X' \pm ZS_E$ where Z is determined by the normal distribution of the curve, considering the percentage confidence limit required. The following Z values should be memorized:

- 68% confidence $\Rightarrow Z = 1.64$
- 95% confidence $\Rightarrow Z = 1.96$
- 99% confidence $\Rightarrow Z = 2.58$

The confidence limit defines where an occurrence X may lie in the range $(X' - ZS_E) < X < (X' + ZS_E)$, given a certain confidence. As you practice solving sampling problems, your confidence in using the formulas will improve.

5.5 Observation and Document Review

Review of source documents will provide useful information about the input, data storage, and output requirements of the software system. This activity could also provide useful information on the processing requirements of the software system. To illustrate, get a hold of an application form at your organization, and attempt to identify information entities (object types) represented on the form. With a little thought, you should be able to identify some or all of the following entities:

- Personal Information
- Family/Kin Contact Information
- Education History
- Employment History
- Professional References
- Extra Curricular Activities

Internal documents include forms, reports and other internal publications.

~~Internal documents include forms, reports and other internal publications,~~
external documents are mainly in the form of journals and other professional publications. These source documents are the raw materials for gaining insights into the requirements of the system, so you want to pay keen attention to them.

With respect to observation, it is always useful to check what you have been told against what you observe and obtain clarification where deviations exist. Through keen observation, the software engineer could gather useful information not obtained by other means.

5.6 Prototyping

In prototyping, the user gets a “trial model” of the software and is allowed to critique it. User responses are used as feedback information to revise the system. This process continues until a software system meeting user satisfaction is obtained (review section 1.4).

The following are some basic guidelines for developing a prototype:

- Work in manageable modules.
- Build prototype rapidly.
- Modify prototype in successive iterations.
- Emphasize the user interface — it should be friendly and meeting user requirements.

There are various types of prototypes that you will find in software engineering. Among the various categories are the following:

Patched-up Prototype or Production Model: This prototype is functional the first time, albeit inefficiently constructed. Further enhancements can be made with time.

Non-operational or Interactive Prototype: This is a prototype that is intended to be tested, in order to obtain user feedback about the requirements of the system represented. A good example is where screens of the proposed system are designed; the user is allowed to pass through these screens, but no actual processing is done.

This approach is particularly useful in user interface design (see [chapter 11](#)).

First of Series Prototype: This is an operational prototype. Subsequent releases are intended to have identical features, but without glitches that the users may identify. This prototype is typically used as a marketing experiment: it is distributed free of charge, or for a nominal fee; users are encouraged to use it and submit their comments about the product. These comments are then used to refine the product before subsequent release.

Selected Features Prototype or Working Model: In this prototype, not all intended features of the (represented) software are included. Subsequent releases are intended to be enhancements with additional features. For this reason, it is sometimes referred to as *evolutionary prototype*. The initial prototype is progressively refined until an acceptable system is obtained.

Throw-away Prototype: An initial model is proposed for the sole purpose of eliciting criticism. The criticisms are then used to develop a more acceptable model, and the initial prototype is abandoned.

5.7 Brainstorming and Mathematical Proof

The methodologies discussed so far all assume that there is readily available information which when analyzed, will lead to accurate capture of the requirements of the desired software. However, this is not always the case. There are many situations in which software systems are required, but there is no readily available information that would lead to the specification of the requirements of such systems. Examples include (but are not confined to) the following:

- Writing a new compiler

- Writing a new operating system
- Writing a new CASE tool, RAD tool, or DBMS
- Developing certain expert systems
- Developing a business in a problem domain for which there is no perfect frame of reference

For these kinds of scenarios, a non-standard approach to information gathering is required. Brainstorming is particularly useful here. A close to accurate coverage of the requirements of an original software product may be obtained through brainstorming: a group of software engineering experts and prospective users come together, and through several stages of discussion, hammer out the essential requirements of the proposed software. The requirements are then documented, and through various review processes, are further refined. A prototype of the system can then be developed and subjected to further scrutiny.

Even where more conventional approaches have been employed, brainstorming is still relevant, as it forces the software engineering team to really think about the requirements identified so far, and ask tough questions to ascertain whether the requirements have been comprehensively and accurately defined.

Mathematical proofs can also be used to provide useful revelations about the required computer software. This method is particularly useful in an environment where formal methods are used for software requirements specification. This approach is often used in the synthesis of integrated circuits and chips, where there is a high demand for precision and negligible room for error. As mentioned in [chapter 1](#), formal methods are not applicable to every problem domain. We will revisit the approach later in the course ([in chapter 12](#)).

5.8 Object Identification

We have discussed six different information-gathering strategies. As mentioned in section 5.1, these strategies are to be used to identify the core requirements of the software system. As mentioned then, one aspect that we are seeking to define is the set of information entities (object types). Notice that the term information entity is used as an alternative to object type. The two terms are not identical, but for most practical purposes, they are similar. An *information entity* is a concept,

object or thing about which data is to be stored. An *object type* is a concept, object or thing about which data is to be stored, and upon which a set of operations is to be defined.

In object-oriented environments, the term *object type* is preferred to information entity. However, as you will more fully appreciate later, in most situations, the software system is likely to be implemented in a *hybrid environment* as an object-oriented (OO) *user interface* superimposed on a *relational database*. This is a loaded statement that will make more sense after learning more about databases ([chapter 10](#)). For now, just accept that we can use the terms *information entity* and *object type* interchangeably in the early stages of software planning.

Early identification of information entities is critical to successful software engineering in the OO paradigm. This is so because your software will be defined in terms of objects and their interactions. For each object type, you want to be able to describe the data that it will host, and related operations that will act on that data. Approaching the software planning in this way yields a number of significant advantages (see [appendix 1](#)); moreover, even if it turns out that the software development environment is not object-oriented, the effort is not lost (in light of the previous paragraph).

[Appendix 4](#) discusses a number of object identification techniques. Among the approaches that have been proposed are the following:

- Using Things to be Modeled
- Using Definitions of Objects, Categories and Types
- Using Decomposition
- Using Generalizations and Subclasses
- Using OO Domain Analysis or Application Framework
- Reusing Individual Hierarchies, Objects and Classes
- Using Personal Experience
- Using the Descriptive Narrative Approach
- Using Class-Responsibility-Collaboration Card
- Using the Rule-of-Thumb Method

Take some time to carefully review [appendix 4](#). To get you adjusted to the idea of object identification, two of the approaches are summarized here.

5.8.1 The Descriptive Narrative Approach

To use the *descriptive narrative approach*, start with a descriptive overview of the software system. For larger systems consisting of multiple subsystems, prepare a descriptive narrative of each component subsystem. From each descriptive overview, identify nouns (objects) and verbs (operations). Repeatedly refine the process until all nouns and verbs are identified. Represent nouns as object types and verbs as operations, avoiding duplication of effort.

To illustrate, the *Purchase Order and Receipt Subsystem* (of an Inventory System) might have the following overview ([Figure 5-6](#)):

Purchase orders are sent to suppliers, requesting inventory items in specific quantities. If a PO is incorrectly generated, it is immediately removed and a new PO generated. The purchase invoice is the official document used to recognize receipt of goods from suppliers. All goods received are accompanied by invoices. Once received, the invoice is recorded. Items received are also recorded and appropriate inventory adjustments made to the inventory item master file. Receipt quantities can be adjusted, but if wrong items are recorded on receipt, or omissions are made, the whole invoice must be removed and re-recorded. When a receipt is correctly recorded, the associated PO status is adjusted.

[Figure 5-6. Descriptive Narrative of Purchase Order and Invoice Receipt Subsystem](#)

From this narrative, an initial list of object types and associated operations can be constructed, as shown below ([Figure 5-7](#)). Further refinement would be required; for instance, additional operations may be defined for each object type (left as an exercise); also, the data description can be further refined (discussed in [chapter 10](#)).

Object Type	Data Description	Operations
Purchase Order	Stores Order Number, Order Date, Supplier, Items Ordered and related Quantity Ordered, etc.	Generate, Remove, Adjust-Status
Supplier	Stores Supplier Code, Supplier Name, Supplier Address, Contact Person, Telephone, E-mail, etc.	Sent-Invoice
Purchase Invoice	Stores Invoice Number, Invoice Date, Related Supplier, Items Shipped and related Quantity Shipped, Invoice Amount, Discount, Tax, etc.	Record, Remove, Adjust-Quantity
Inventory Item	Stores Item Code, Item Name, Item Category, Quantity on Hand, Last Purchase Price, etc.	Adjust-Inventory

[Figure 5-7. Object Types and Operations for Purchase Order and Invoice Receipt Subsystem](#)

5.8.2 The Rule-of-Thumb Approach

As an alternative to the descriptive narrative strategy, you may adopt an intuitive approach as follows: Using principles discussed earlier, identify the main information entities (object types) that will make up the system. Most information entities that make up a system will be subject to some combination of the following basic operations:

- **ADD:** Addition of data items
- **MODIFY:** Update of existing data items
- **DELETE:** Deletion of existing data items
- **INQUIRE/ANALYZE:** Inquiry and/or analysis on existing information
- **REPORT/ANALYZE:** Reporting and/or analysis of existing information
- **RETRIEVE:** Retrieval of existing data
- **FORECAST:** Predict future data based on analysis of existing data

Obviously, not all operations will apply for all object types (data entities); also, some object types (entities) may require additional operations. The software engineer makes intelligent decisions about these exceptions, depending on the situation. Additionally, the level of complexity of each operation will depend to some extent on the object type (data entity).

In a truly OO environment, the operations may be included as part of the object's services. In a hybrid environment, the information entities may be implemented as part of a relational database, and the operations would be implemented as user interface objects (windows, forms, etc.).

5.9 Summary and Concluding Remarks

Here is a summary of what we have covered in this chapter:

- It is important to conduct a research on the requirements of a software system to be developed. By so doing, we determine

the synergistic interrelationships, information entities, operations, business rules, and security mechanisms.

- In conducting the software requirements research obtaining answers to questions commencing with the words WHAT, WHERE, HOW, WHEN and WHY is very important.
- Information gathering strategies include interviews, questionnaires and surveys, sampling and experimenting, observation and document review, prototyping, brainstorming and mathematical proofs.
- The interview is useful when the information needed must be elaborate, or clarification on various issues is required. The interview also provides an opportunity for the software engineer to win the confidence and trust of clients. In preparing to conduct an interview, the software engineer must be thoroughly prepared, and must follow well-known interviewing norms.
- A questionnaire is viable when any of the following situations hold: A small amount of information is required of a large population; the time frame is short but a vast area (and/or dispersed population) must be covered; simple answers are required to a number of standard questions. The software engineer must follow established norms in preparing and administering a questionnaire.
- Sampling is useful when the information required is of a quantitative nature or can be quantified, no precise detail is available, and it is not likely that such details will be obtained via other methods. The software engineer must be familiar with various sampling techniques, and know when to use a particular technique.
- Review of source documents will provide useful information about the input, data storage, and output requirements of the software system. This activity could also provide useful information on the processing requirements of the software system.
- Prototyping involves providing a trial model of the software for user critique. User responses are used as feedback

information to revise the system. This process continues until a software system meeting user satisfaction is obtained. The software engineer should be familiar with the different types of prototypes.

- Brainstorming is useful in situations in which software systems are required, but there is no readily available information that would lead to the specification of the requirements of such systems. Brainstorming involves a number of software engineers coming together to discuss and hammer out the requirements of a software system.
- Mathematical proof is particularly useful in an environment where formal methods are used for software requirements specification.
- One primary objective of these techniques is the identification and accurate specification of the information entities (or object types) comprising the software system.

Accurate and comprehensive information gathering is critical to the success of a software engineering venture. In fact, the success of the venture depends to a large extent on this. Your information gathering skills will improve with practice and experience.

In applying these techniques, the software engineer will no doubt gather much information concerning the requirements of the software system to be constructed. How will you record all this information? If you start writing narratives, you will soon wind up with huge books that not many people will bother to read. In software engineering, rather than writing voluminous narratives to document the requirements, we use unambiguous notations and diagrams (of course, you still need to write but not as much as you would without the notations and diagrams). The next chapter will discuss some of these methodologies.

5.10 Review Questions

1. Why is information gathering important in software engineering?

2. Identify seven methods of information gathering that are available to the software engineer. For each method, describe a scenario that would warrant the use of this approach, and provide some basic guidelines for its application.
3. Suppose that you were asked to develop an inventory management system (with point-of-sale facility) for a supermarket. Your system is required to track both purchase and sale of goods in the supermarket. Do the following:
 - Prepare a set of questions you would have for the purchasing manager.
 - Prepare a set of questions you would have for the sales manager.
 - Apart from interviews, what other information gathering method(s) would you use in order to accurately and comprehensively capture the requirements of your system? Explain.
4. Suppose that you are working for a software engineering firm that is interested in developing a software to detect certain types of cancer, based on information fed to it. Your software will also suggest possible treatment for the cancer diagnosed. You are given a list of twelve physicians who are cancer experts; they will form part of your resource team. Answer the following questions:
 - What type of software would you seek to develop and why?
 - What methodology would you use for obtaining critical information from the cadre of physicians?
 - Construct an information gathering instrument that you would use in this project.
5. The manager of seaport wants to be at least 95% certain that there is a serious bug in the current dock scheduling system, before considering its replacement. He contracts a software engineer to advise him. After sampling forty (40)

docking schedules, the software engineer runs each through the system and obtains a docking error factor. He summarizes his findings as follows:

Sample Size 40

Margin of error ... +/- 1 unit

Standard deviation ... 2.795

Mean docking error factor ... 10 units

Note: 95% => $Z = 1.96$ 99% => $Z = 2.58$

- a. Should the manager replace the dock scheduling system?
 - b. What docking error factor will yield a 99% confidence of the presence of a bug in the system?
6. What type of prototype would you construct for the following:
- The cancer diagnosis system of question 4
 - A new compiler that you hope to obtain feedback on
 - A user workgroup designed to elicit useful information for the requirements of a financial management system

5.11 References and/or Recommended Readings

[Daniel, 1989] Daniel, Wayne, and Terrel, James. *Business Statistics for Management and Economics* 5th ed. Boston, MA: Houghton Mifflin Co., 1989.

[DeGroot, 1986] DeGroot, Morris H. *Probability and Statistics* 2nd ed. Reading, MA: Addison-Wesley, 1986.

[Harris, 1995] Harris, David. *Systems Analysis and Design: A Project Approach*. Fort Worth, TX: Dryden Press, 1995. See [chapters 3, 4](#).

[Kendall, 1999] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999. See [chapters 4 – 8](#).

[Long, 1989] Long, Larry. *Management Information Systems*. Eaglewood Cliffs, NJ: Prentice Hall, 1989. See [chapter 13](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapter 4](#).

[Sommerville, 2001] Sommerville, Ian. *Software Engineering* 6th ed. Reading, MA: Addison-Wesley, 2001. See [chapter 8](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See [chapter 7](#).

[Van Vliet, 2000] Van Vliet, Hans. *Software Engineering* 2nd ed. New York, NY: John Wiley & Sons, 2000. See [chapter 9](#).

CHAPTER 6



Communicating Via Diagrams

In the previous chapter, we discussed information gathering. Remember, as a software engineer, you do not gather information (or anything for that matter) just for the sake of doing so. Rather, there must be a purpose, and as pointed out earlier, your objective is the preparation of the project's second major deliverable — the requirements specification. As you will soon see, this deliverable can be quite bulky, particularly if the software system is quite complex and/or large. In software engineering, we do not like unnecessary fluff; we promote comprehensive but succinct coverage.

Comprehensive coverage and brevity are often not easy to achieve since they are to some extent mutually contradictory. However, to assist in the pursuit of these sometimes-conflicting ideals, we use diagrams. This chapter provides you with a broad overview of various diagramming techniques that are used in documenting the requirements of computer software. It proceeds under the following captions:

- Introduction
- Traditional System Flow Charts
- Procedure Analysis Chart
- Innovation: Topology Charts
- Data Flow Diagrams
- Object Flow Diagram
- Other Contemporary Diagramming Techniques
- Program Flow Chart
- Summary and Concluding Remarks

6.1 Introduction

In the preparation of the requirements specification, the software engineer invariably employs various diagramming techniques in order to convey information about the software. Diagrams provide graphic representation of the flow of information, as well as the inter-relationships among system resources. In developing diagrams, the software engineer uses predefined symbols that have established meanings.

Among the many advantages of system diagrams are the following:

- They aid in the illustration of logical inter-relationships of various components of a system.
- They help in the development of system logic since they show from start to finish, various conditions, actions and data storage in the system.
- They are traceable.
- They can help to identify bottlenecks and weaknesses in the system.
- In the object-oriented paradigm (particularly with OO-CASE tools), diagrams are executable.

The diagramming techniques discussed in this chapter are drawn from the traditional function-oriented (FO) software engineering paradigm, as well as the more contemporary object-oriented (OO) paradigm. While contemporary products tend to be designed based on the OO paradigm, there are many legacy systems that still abound. An understanding of both approaches is therefore imperative for the keen software engineer. [Figure 6-1](#) provides a list of some commonly used diagrams and the software engineering paradigm(s) (OO or FO) to which they apply. Some of these diagrams will be discussed in this chapter; others will be discussed in more appropriate sections later in the course.

Diagramming Technique	SE Paradigm
Information Oriented Flow Chart	FO
Process Oriented Flow Chart	FO
HIPO Chart	FO
Information Topology Chart (ITC)	OO and FO
User Interface Topology Chart (UITC)	OO and FO
Object Flow Diagram (OFD)	OO
Fern Diagram	OO
Object Structure Diagram (OSD)	OO
Object Relationship Diagram (ORD)	OO
Entity Relationship Diagram (ERD)	FO and OO
State Transition Diagram (STD)	FO and OO
Finite State Machine (FSM)	FO and OO
Procedure Analysis Chart	FO and OO
Data Flow Diagram (DFD)	FO
Program Flow Chart	FO
Warnier-Orr Diagram	FO and OO
Event Diagram or Activity Diagram	OO
Collaboration Diagram	OO
Unified Modeling Language (UML) Diagram	OO
Decision Table & Decision Tree	FO and OO
Gantt Chart	FO and OO
PERT Diagram	FO and OO

Figure 6-1. Commonly Used Diagramming Techniques

6.2 Traditional System Flowcharts

Traditional system flow charts include information-oriented flow charts, HIPO charts and process-oriented flow charts. [Figure 6-2](#) shows symbols used for these traditional system flow charts. This figure provides a good opportunity to clarify a few terms that are commonly used in software requirements specification:

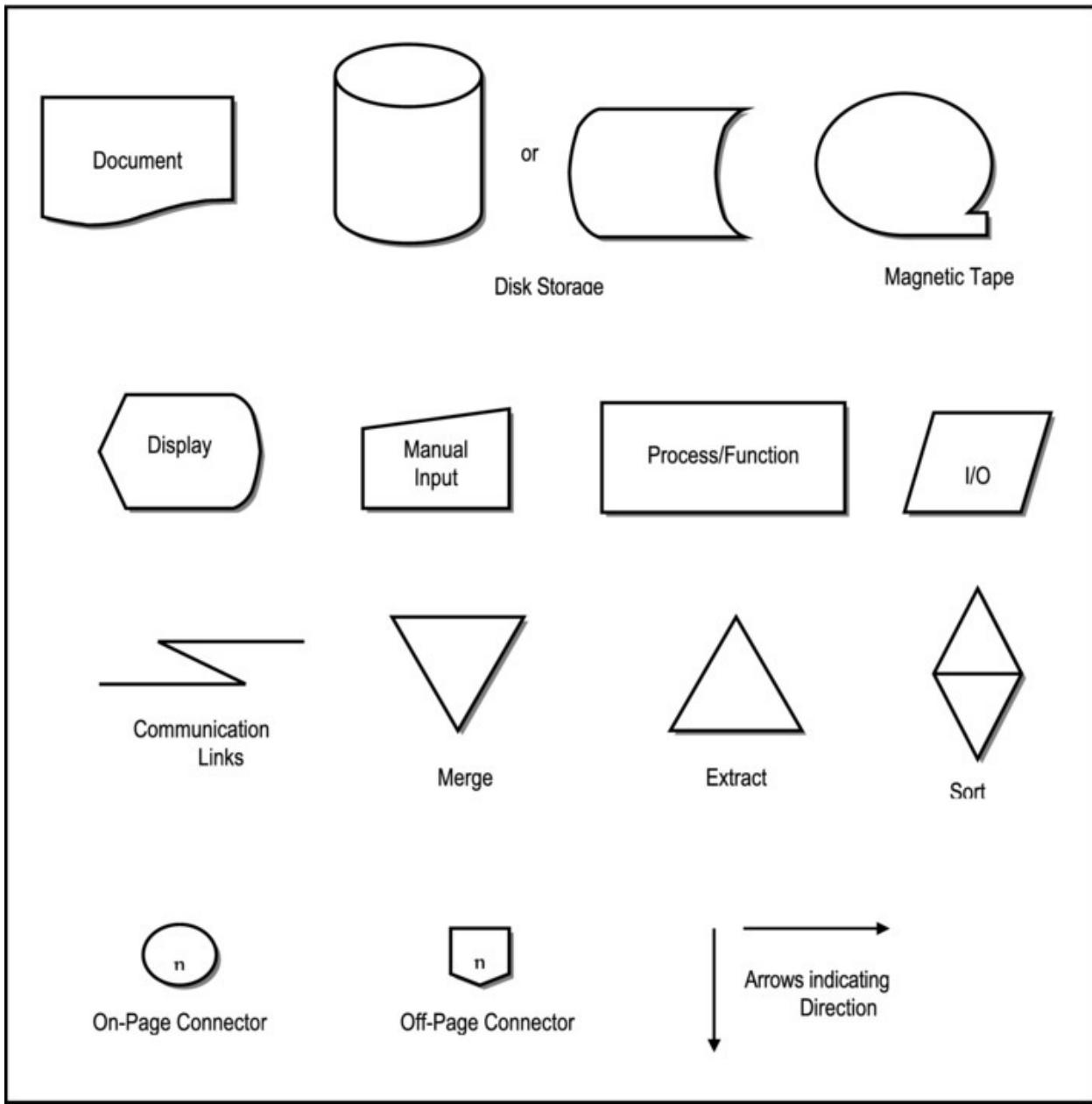


Figure 6-2. Traditional System Flowchart Symbols

Disk or Data Storage: This represents the storage of a data entity (items) to an appropriate storage device (typically magnetic disk or optical disk). As you will learn later in the course, these entities are comprised of data elements (also called attributes). However, at this stage, we do not concern ourselves with data elements; only the data storages.

Process or Function or Operation: Some texts will distinguish between process and function. Such distinctions are frivolous at best, and confusing at worst; this course makes no such distinction; rather, the terms will be used as

synonyms (in the context of the FO paradigm). A process (or function) describes a set of related activities that can be conveniently summarized in the descriptive name given to the process. Functions eventually translate to actual programs that will be written as part of the software system. In the OO paradigm, the term operation is preferred to function or process. Throughout this text, this is the preferred term, except when dealing with traditional (function oriented) techniques. Finally, it must be borne in mind that operations translate to actual methods of classes, or classes with constituent methods that will be written at development time. For further clarification, please review [appendix 2](#).

Subsystem: A subsystem is an independent (or almost independent) component of a software system. The subsystem may operate as part of a larger system, or on its own as an independent system.

Module: A module is a subservient component of a system or subsystem. The module does not operate on its own.

6.2.1 Information-Oriented Flowchart

The *information-oriented flowchart* (IOF) has the following characteristics:

- It traces the flow of information through the organization.
- It is usually grid structure.
- It uses mainly the document symbol.
- It is normally accompanied by a narrative that describes the information flow steps.

[Figure 6-3](#) shows an example on an IOF. Notice that in this example, only the document symbol is used to show the flow of information across different departments in the organization. This is typical of IOFs.

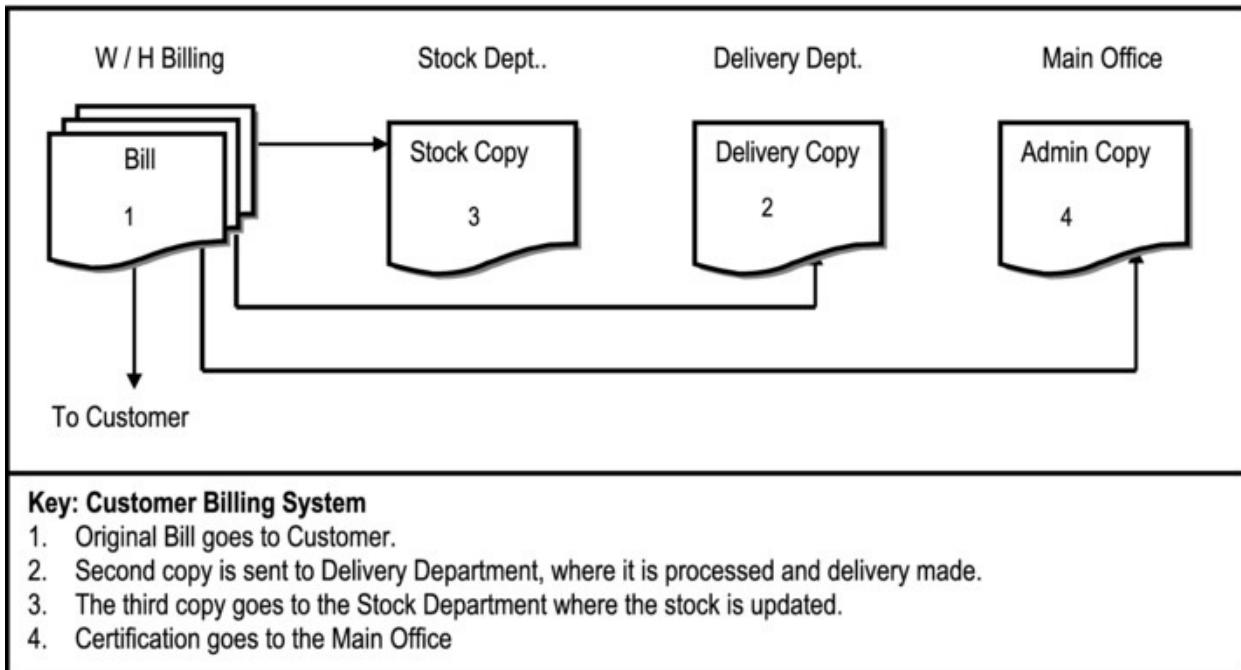


Figure 6-3. Example of Information-Oriented Flowchart

6.2.2 Process-Oriented Flowchart

The *process-oriented flowchart* (POF) has the following characteristics:

- It traces the processing of information throughout the organization.
- It may be highly summarized or fairly detailed.
- It uses mainly the I/O symbol, the process symbol, the document symbol, and the storage symbol.
- It is normally accompanied by a descriptive narrative.
- It is useful at the analysis, system specification and design stages.

Figure 6-4 shows an example of a POF. Notice from the example that the POF conveys a number of important pieces of information about the software system as mentioned below:

- The inputs to the system

- The important processes comprising the system
- The important data storages comprising the system (these translate to information entities in the underlying database)
- The critical outputs from the system

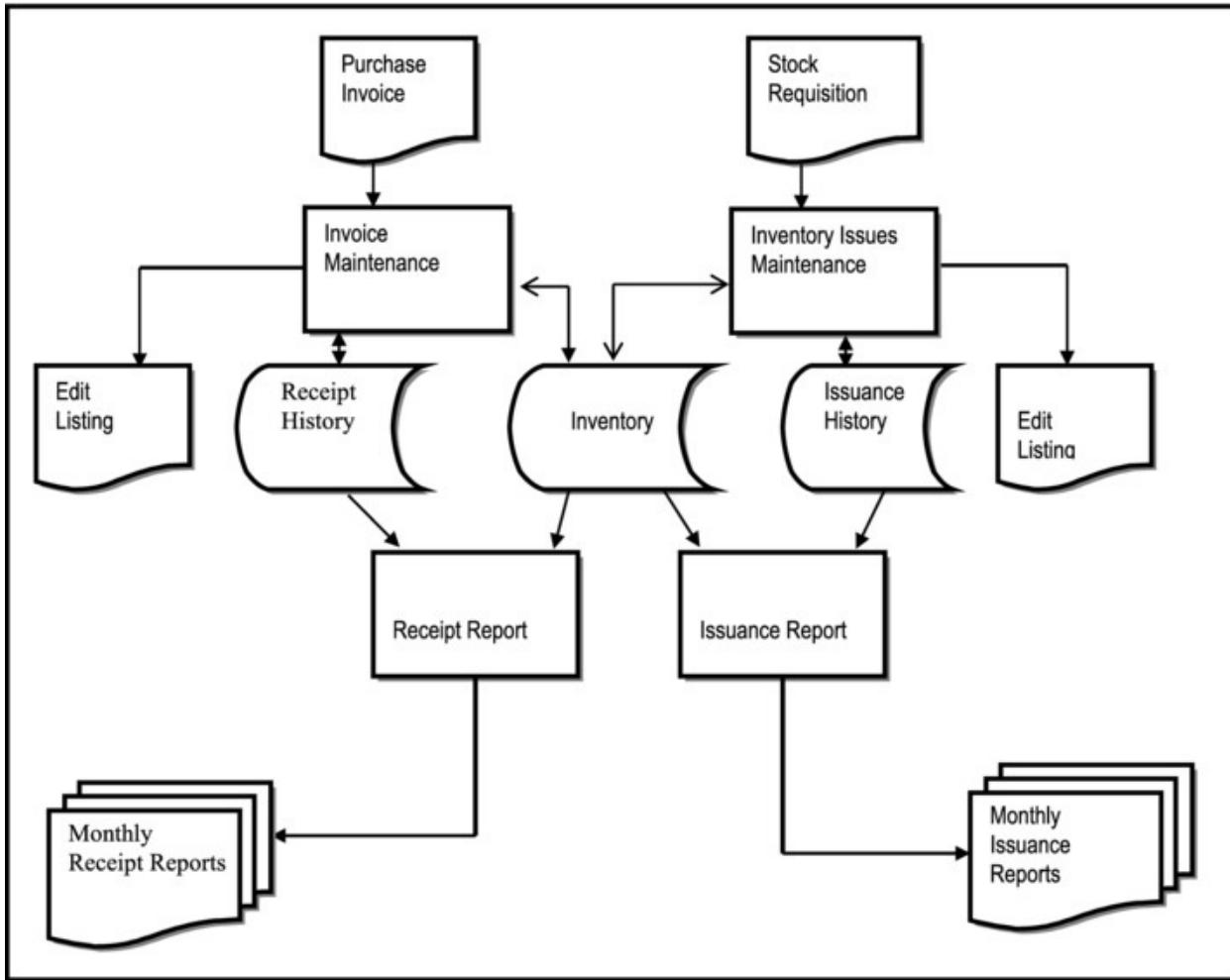


Figure 6-4. Process-Oriented Flowchart for (part of an) Inventory Management System

6.2.3 Hierarchy- Input–Process–Output Chart

The *hierarchy-input-process-output* (HIPO) chart has the following characteristics:

- It presents the system and its main functional components in a hierarchical manner, so that relationships among them can

be easily depicted.

- The name of the software system is at level-1; the second and/or intermediate levels contain major functional components of the system (subsystems, modules, and eventually functions/processes); for the final level, each function is broken down into component activities.
- A second IPO chart can show more details about each functional module, outline inputs, processing steps and outputs. This is useful at the design phase.

Figure 6-5 illustrates a HIPO chart for the earlier mentioned inventory management system. Notice that on the HIPO chart, system functions are usually indicated at the lowest level of the hierarchy.

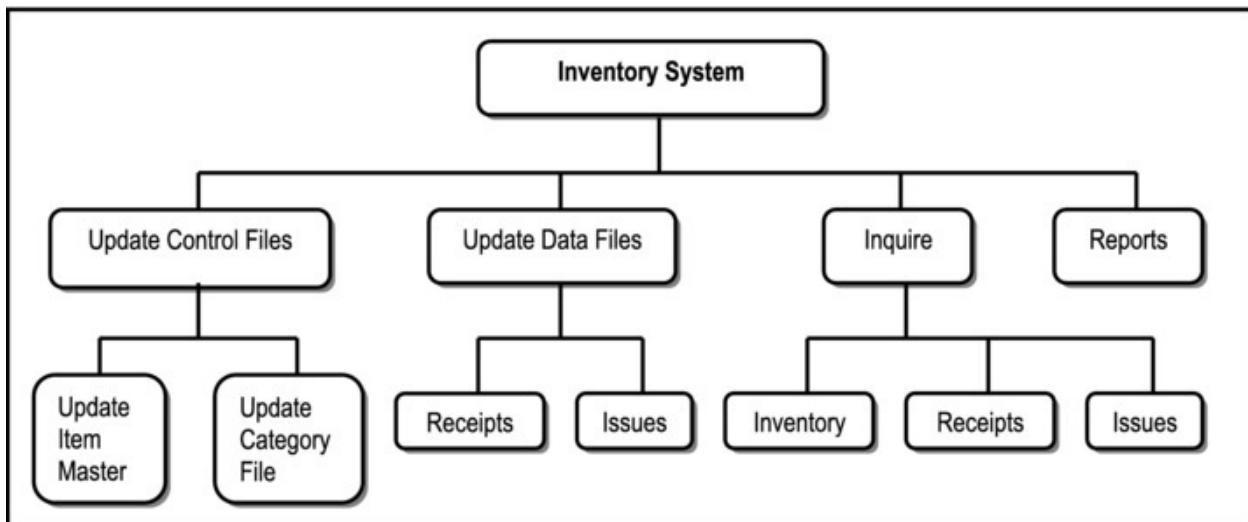


Figure 6-5a. Example of a HIPO Chart

Function: Update Item Master Operation Description: Allows maintenance of the information stored in the Inventory Master File.		
Input	Processing	Output
Inventory Items, Inventory Categories	1. Accept Item Number 2. If this is a new Item Number, allow addition of a new inventory item	Edit Listing, Inventory Master File
	3. If this is a preexisting Item Number, find out if the user desires modification or deletion	
	4. If modification is chosen, allow modification of the inventory record; otherwise, allow deletion of the employee record	

Figure 6-5b. Example of an IPO Chart

6.3 Procedure Analysis Chart

The *procedure analysis chart* (PAC) is used to conduct process analysis in the organization. Its main objective is to identify bottlenecks and solve them by any of the following strategies:

- Eliminating delays
- Merging processes
- Eliminating redundant processes
- Introducing new processes if necessary

The procedure analysis chart is particularly useful in process re-engineering in the organization. [Figure 6-6](#) provides an example. From the chart, you will observe that there are five types of actions that are studied: activity, transportation, inspection, delay and storage. The summary provides the total time by each type of action, for that particular job function. By examining the chart, the software engineer can make recommendations as to whether improvements are needed in that particular functional area of the system. For this reason, PACs are extremely useful in business process reengineering (BPR).

Summary	Present		Proposed		Difference	
	No.	Time	No.	Time	No.	Time
# Activities	7	7.42				
# Transportations	4	0.47				
# Inspections	2	3.05				
# Delays	1	0.02				
# Storages	1	0.3				

Note: 1. Time given in Minutes
2. Distance given for transportation in meters (second figure in transportation column).

Action	Activity	Transportation	Inspection	Delay	Storage	Action Change (combine, delete, improve)
Select Next Sale Order	0.02					
Examine Credit Request			3.00			
Calculate Sale Amount	0.50					
Find Customers File	0.30					
Record Customers Balance	0.10					
Obtain Supervisor's Approval		0.12, 20m				
Note Unprocessed Memos	5.00					
Return to Desk		0.25, 40m				
Calculate New Customer Bal.	0.50					
Compare New Bal. to Limit			0.05			
Approve or disapprove	1.00					
Place 3 Copies of Order in Out-tray				0.02		
Take Copy to Customer File		0.05, 15m				
File Customer Information					0.30	
Return to Desk		0.05, 15m				

Figure 6-6. Procedure Analysis Chart

6.4 Innovation: Topology Charts

The following two topology charts have been proposed by the current writer and may be employed in both (OO and FO) paradigms of software engineering.

6.4.1 Information Topology Chart

The *information topology chart* (ITC) shows information levels of the software system in a top-down manner — the system is at the highest level and data elements at the lowest level. It presents information to be managed in the system in a logical and modular way, and therefore allows for easy analysis and identification of omissions or redundancies.

The ITC is particularly useful in providing a global view of the system, including all significant components. It is useful for analysis and specification, as well as the design phases of the SDLC. The technique differs from the HIPO chart in the following way: The HIPO chart is a functional representation of processes in a system. On the other hand, the ITC is a conceptual representation of component information entities (object types) of the system.

The ITC also differs from other techniques such as fern diagram, object-relationship diagram, and object flow diagram. A comparison of the ITC with these methodologies is available in reference [Foster, 1999]. This information has been excluded because it is not considered necessary for this course. However, it is useful to mention some benefits of the approach:

- The ITC is a useful design and documentation aid.
- The technique is easy to learn, involving minimal use of symbols.
- The technique is useful in conceptualizing (the entire) system scope.
- The technique is useful in illustrating how information (object types) will be managed.
- The technique is useful applicable in the FO paradigm as well as OO paradigm.

The ITC is more useful in OOSE, but can also be used in the traditional approach. [Figure 6-7](#) illustrates a portion of an ITC for a generic *College/University Administrative Information System* (CUAIS). From this diagram, it can be seen that the CUAIS is an integrated software system consisting of nine independent but interrelated subsystems. Software systems of this sort belong to a family called *enterprise resource planning* (ERP) systems, which in turn belongs to a larger family called *management support systems*.

(MSSs). Each subsystem is responsible for the management of various information entities, some of which have been included in the diagram. [Appendix 10](#) includes additional ITC illustrations.

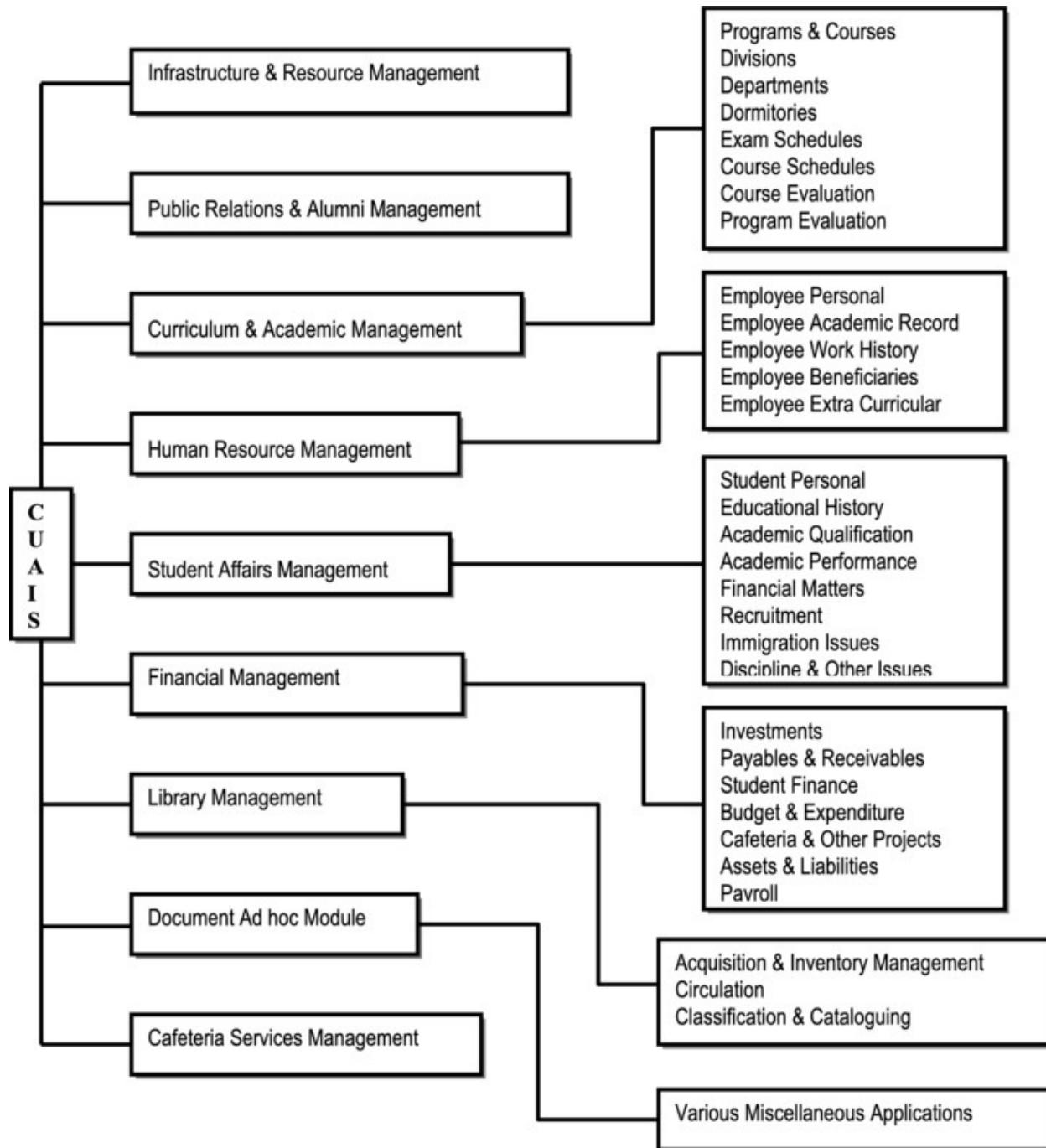


Figure 6-7. Partial Information Topology Chart for the CUAIS Project

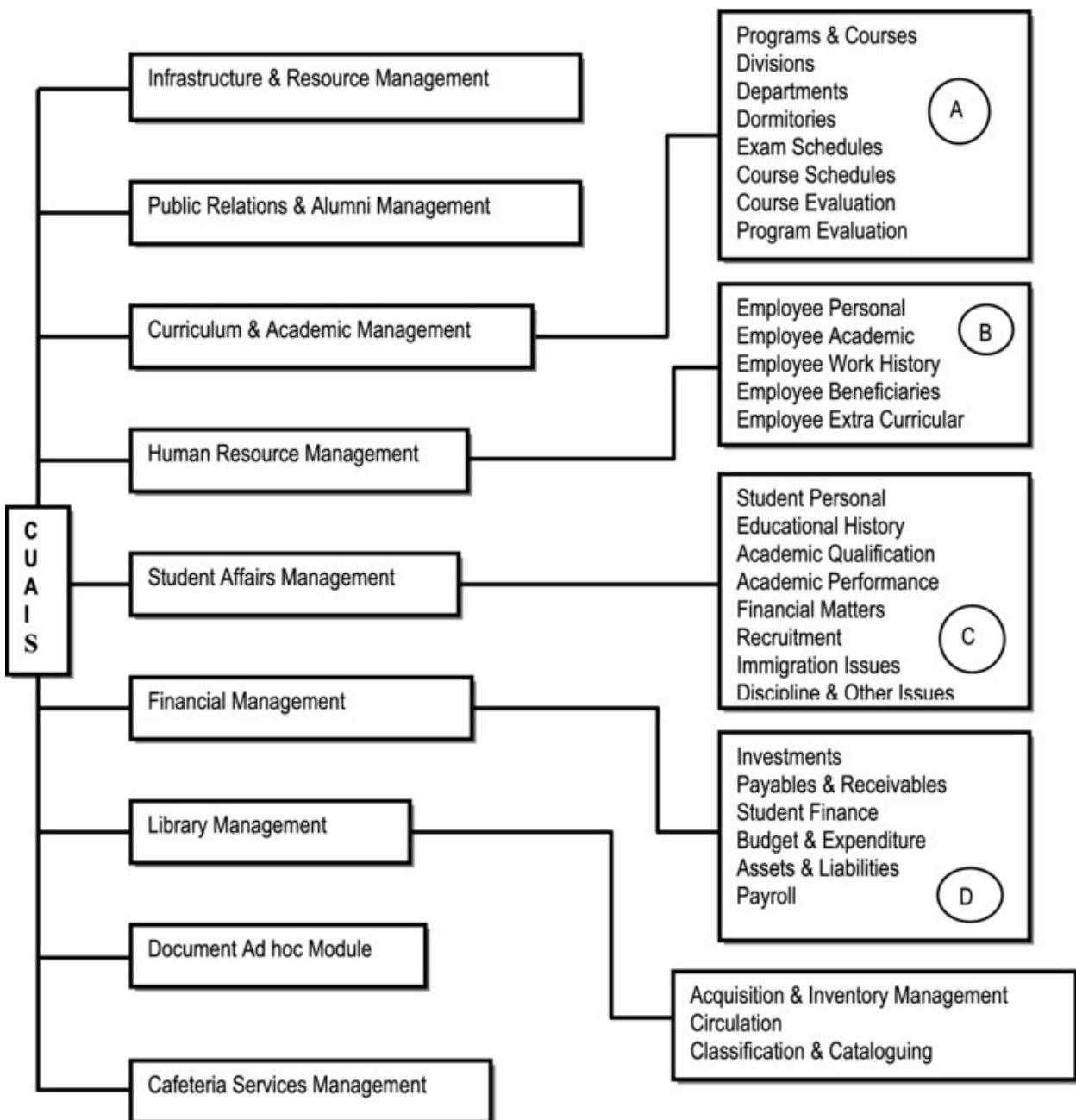
6.4.2 User Interface Topology Chart

----- r - o -----

The *user interface topology chart* (UITC) is logically constructed from the ITC, and is comparable to Schneideman's *Object-Action Interface* (OAI) model for user interfaces [Schneiderman, 2005].

It shows the operational levels of the system in a top-down manner: the system is represented at the highest level; subsystems (may) appear at the intermediate levels; actual operations are represented at the lowest level. It is similar to a HIPO chart, except that it favors an OO approach to software design.

The UITC presents operations of the system in a logical manner, showing interrelationships, and how they fit in the overall system architecture. It also presents the end user with a panoramic perspective of the entire system. Hence, as the name suggests, it is useful in portraying a blown out static picture of the (menu driven or graphical) user interface of the system. [Figure 6-8](#) illustrates a partial UITC for the CUAIS project. Additional illustrations of UITCs are available in [appendix 10](#).



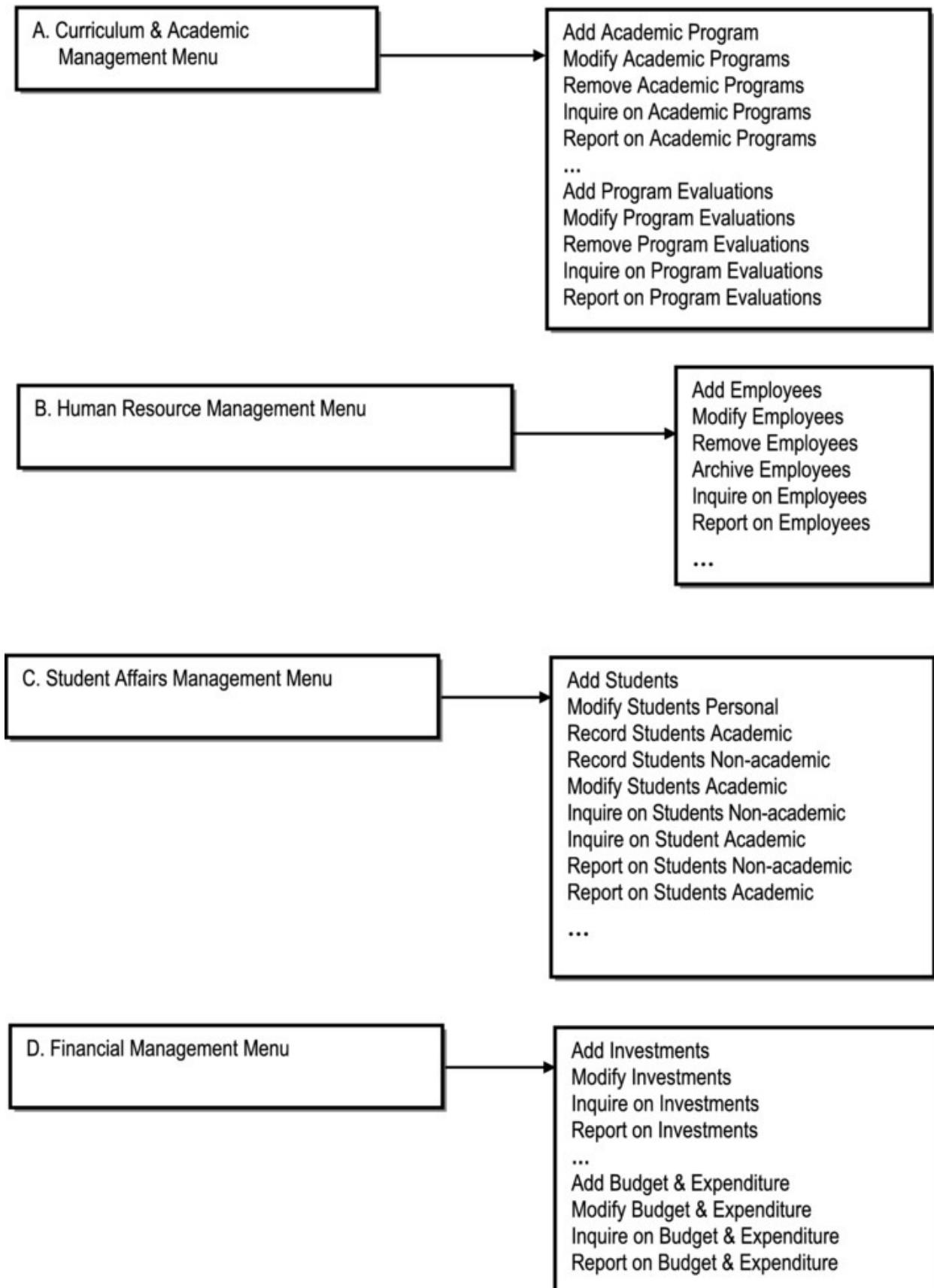


Figure 6-8. Partial User Interface Topology Chart for the CUAIS Project

6.5 Data Flow Diagrams

A *data flow diagram* (DFD) reflects the main information storage, information flow and processes of a software system all in one. It has relevance in the FO paradigm, as a useful analysis technique for representing the logic underlying a system. [Figure 6-9](#) shows the symbols used. Notice the similarity between the process and the data storage symbols. In the original proposals for DFD, the process symbol is a rectangle with rounded corners, and a data storage symbol is a rectangle with one side opened. In the interest of simplicity, these intricacies have been relaxed.

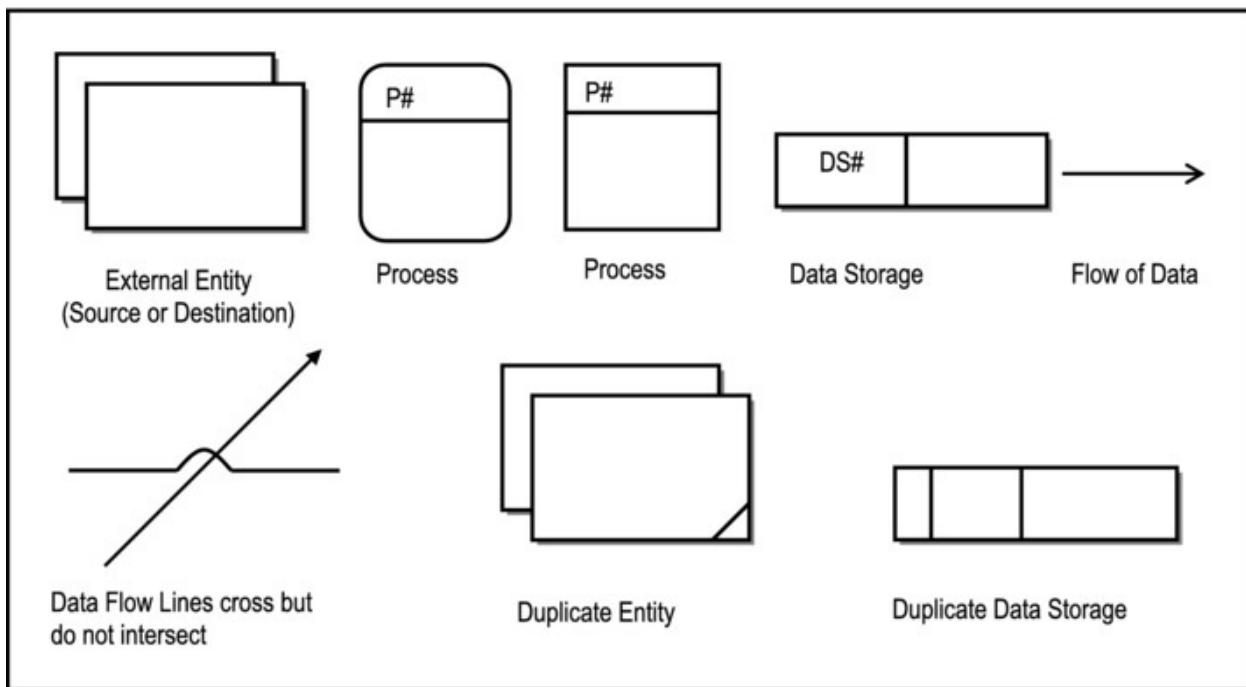


Figure 6-9. Symbols Used in DFD

The DFD provides the following advantages:

- It enhances understanding of the interrelatedness of systems and subsystems.
- It is an effective tool in communicating to users.

In drawing a DFD, the following conventions are employed:

- Indicate process number at the top of the process rectangle. Use decimals to indicate sub-processes.
- Indicate data store number at the left of the data store rectangle.
- Label entities, processes and data stores on the inside of respective rectangles.
- Develop DFD in a top-down manner; level-0 to level-n. Explode from one level to the next in order to reveal more detail about the processes. At each level, identify external entities, processes, data flows and data storages.

By way of illustration, let us revisit the inventory management system (IMS) introduced earlier in the chapter, and take a closer look. Consider the case where the manager of a supermarket, variety store, or auto-store desires such a system that will allow him/her to effectively keep track of products needed to keep the store in operation. In each of these cases, the system implementation will have several points of similarities with the other two scenarios, and a few points of differences. Let us for the moment concentrate on some of the similarities. This illustration does not attempt to cover all aspects of the IMS; rather it covers some of the salient features:

- Two obvious external entities that are crucial to the implementation of the software system are suppliers of inventory items and the customers who purchase from the business.
- [Figure 6-10](#) provides a list of some important processes and data storage objects that would comprise the IMS (this is not a comprehensive list). As you view this, remember that the storage objects typically translate to information entities in the underlying database.

Process	Description
P01: Prepare Purchase Orders	Facilitates the generation of purchase orders that are sent to suppliers.
P02: Modify Purchase Orders	Facilitates modification of purchase orders before sending them off to suppliers.
P03: Remove Purchase Orders	If a purchase order was incorrectly generated, this process facilitates its removal.
P04: Query Purchase Orders	Facilitates the querying of one or more purchases orders.
P05: Receive Purchase Invoices	Facilitates the recording of purchase invoices received from suppliers.
P06: Modify Purchase Invoices	Facilitates modification of purchase invoices where errors might have been made.
P07: Remove Purchase Invoices	Facilitates the removal of incorrectly recorded purchase invoices.
P08: Query Purchase Invoices	Facilitates the querying of one or more purchases invoices.
P09: Pay Creditors	Facilitates recording the payment of funds to suppliers and other creditors (for instance the bank, etc).
P10: Add New Inventory Items	Facilitates the addition of new inventory items to the system.
P11: Modify Inventory Items	Facilitates modification of inventory item(s).
P12: Remove Inventory Items	Facilitates the removal of incorrectly entered inventory items.
P13: Query Inventory	Facilitates querying one or more inventory items.
P14: Generate Sale Invoices	Facilitates modification of sale invoices before they are sent to customers
P15: Modify Sale Invoices	Facilitates the generation of receipts
P16: Remove Sale Invoices	Facilitates the removal of sale invoices that were incorrectly generated.
P17: Query Sales	Facilitates querying one or more sale activities.
P18: Receive Payments	Facilitates the recording of payments received for goods sold (on credit).
P19: Perform Cash Sales	Facilitates the processing of cash sales.

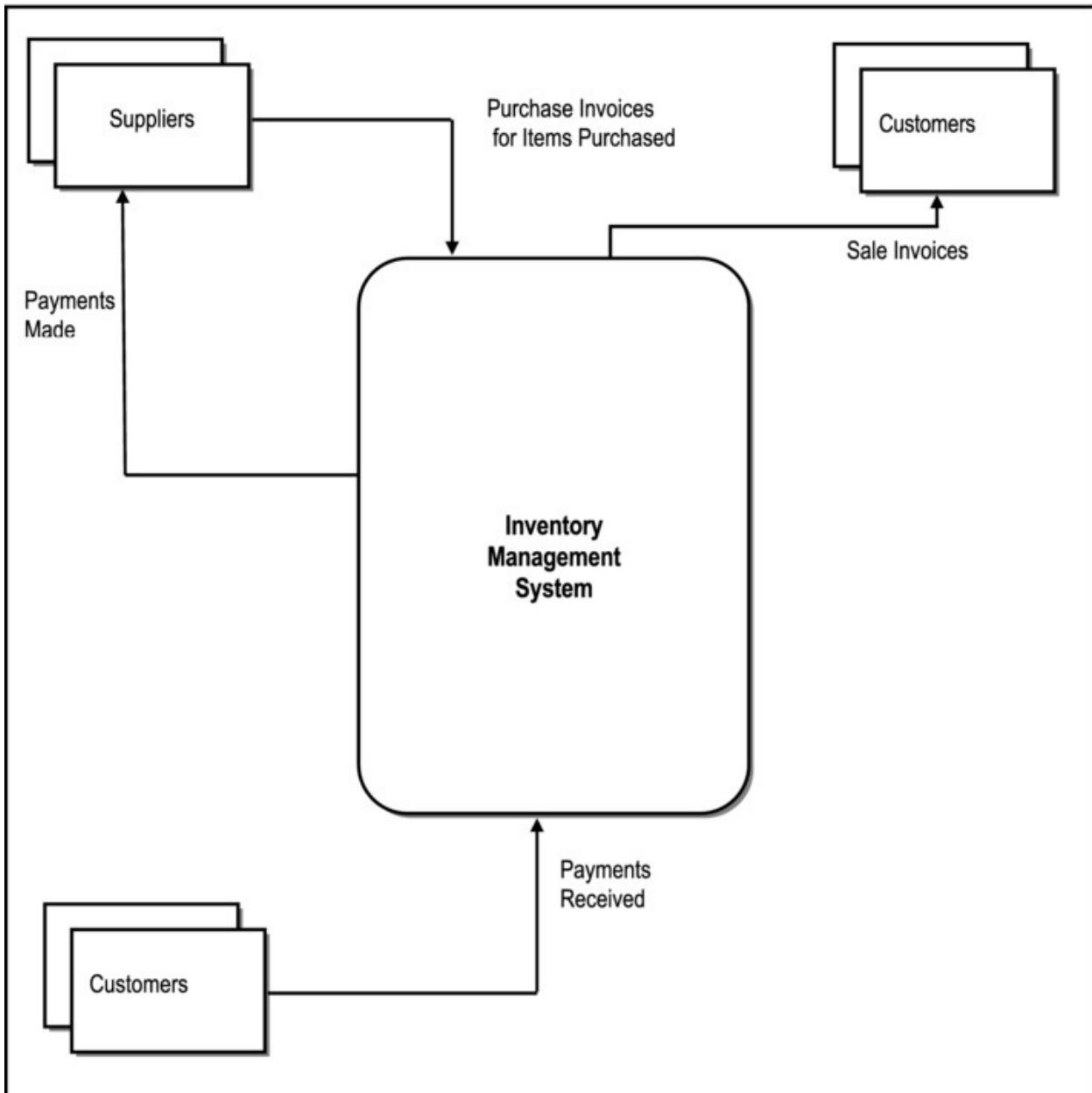
Figure 6-10a. Important Processes in an Inventory Management System

Data Storage	Description
D01: Inventory Items	Stores a record of each inventory item; includes category, on hand quantity, purchase price, sale price, and other related data.
D02: Purchase Orders	Stores records of each purchase order generated; includes order number, supplier sent to, order date, items ordered and related quantities, etc.
D03: Purchase Invoices	Stores records of all purchases made; includes invoice date, invoice number, related supplier and purchase order, items received and related quantities and amounts, etc.
D04: Accounts Payable	Stores information on amounts owed to creditors.
D05: Payments Made	Stores information on payments made to creditors.
D06: Sale Invoices	Stores records of all sales made; similar to D03.
D07: Accounts Receivable	Stores information on amounts owed to the store by customers.
D08: Payments Received	Stores information on amounts paid to the store; includes cash sales.
D09: Customers	Stores information on customers of the store
D10: Suppliers	Stores information on suppliers of the store.

Figure 6-10b. Important Data Storage Objects in an Inventory Management System

Figure 6-11 shows a level 0 DFD called a *context diagram*. Notice that at this level, the system is perceived as a large process. Figure 6-12 shows a (partial) level-1 DFD of the system, highlighting processes P01, P05, P09, P11, P14, P18 and P19 (see Figure 6-10). These were chosen because they represent the most critical processes in the system. You would then have a choice as to

how to continue refining your requirements: for each of the processes specified in your level-1 diagram ([Figure 6-12](#)), you could specify a level-2 DFD as illustrated in [Figure 6-13](#), a program flowchart (see section 6.8), an IPO chart (review section 6.2.3), a Warnier-Orr diagram (discussed in [chapter 12](#)), an activity diagram (discussed in [chapter 12](#)), or an extended operation specification (discussed in [chapter 12](#)).



[Figure 6-11.](#) Context Diagram for the Inventory Management System

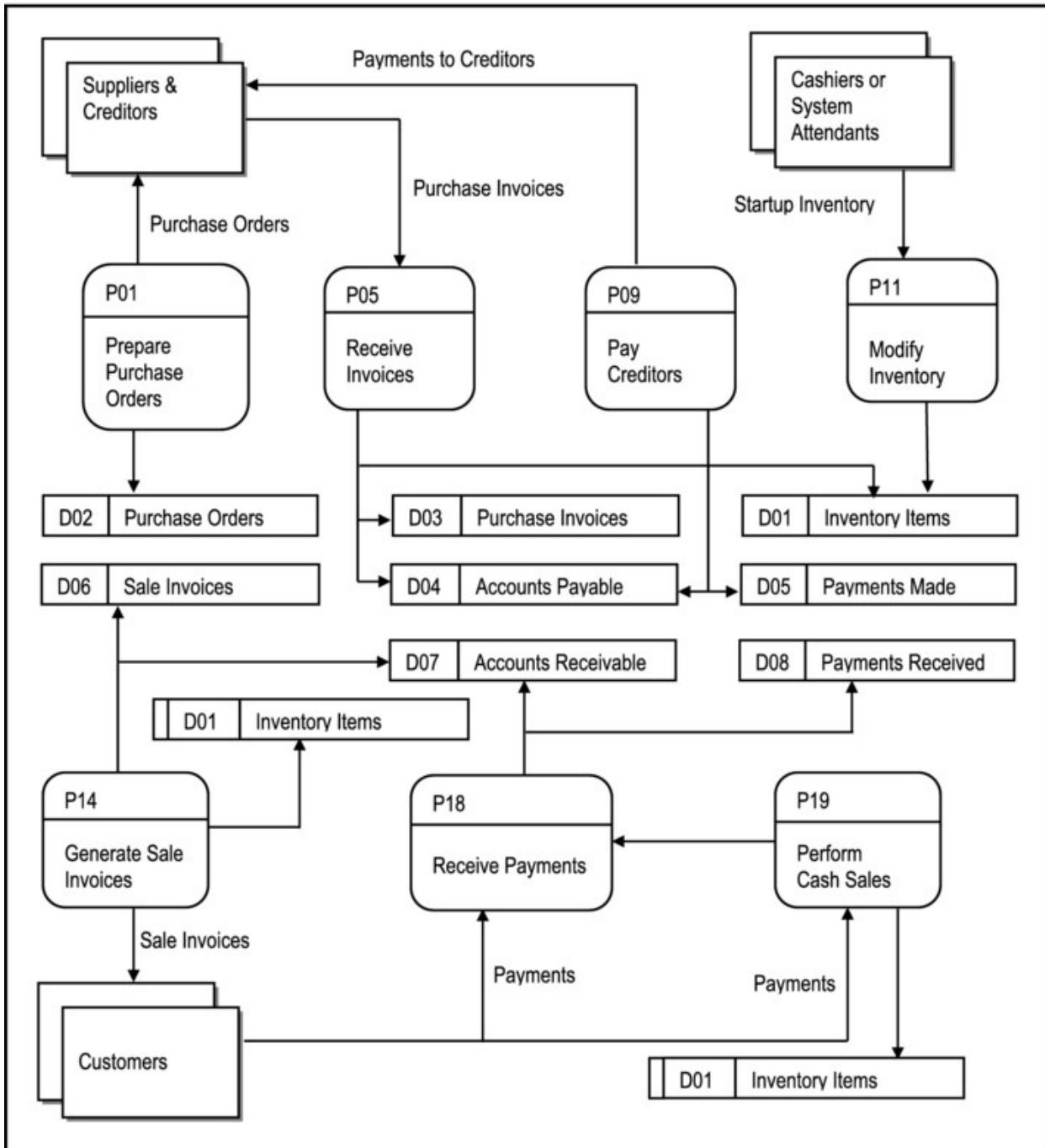


Figure 6-12. Level-1 DFD for the Inventory Management System

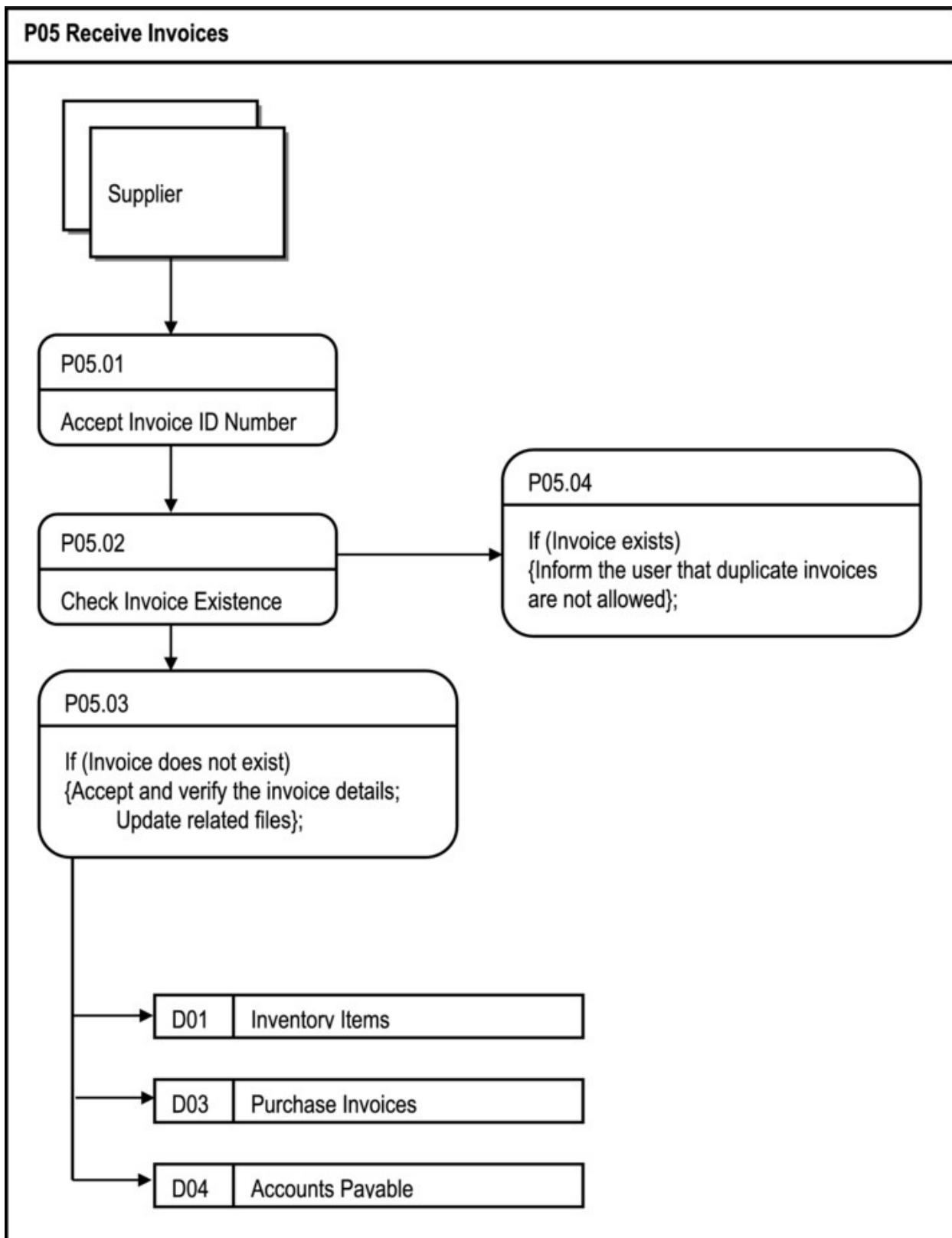
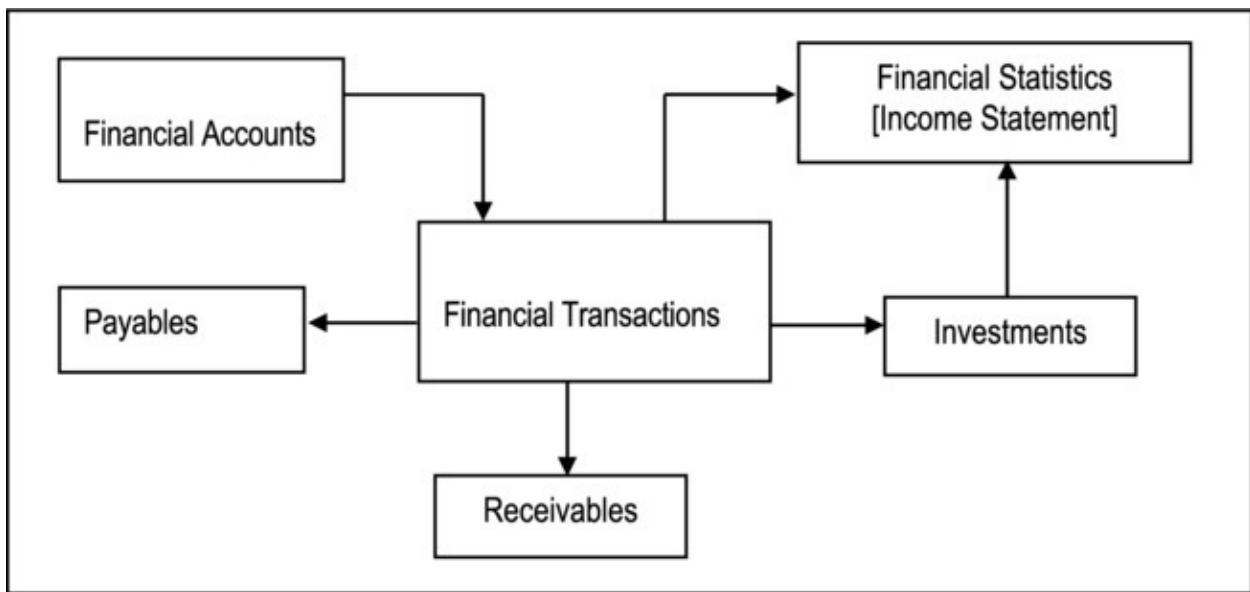


Figure 6-13. Level-2 DFD for Process P05 of the Inventory Management System

6.6 Object Flow Diagram

An *object flow diagram* (OFD) is typically used in OOSE to show how major software system components (super classes or subsystems) communicate with one another. It is normally used at a high level of system specification. Here, it has similarities to a DFD level-0 and an ITC at the highest level. [Figure 6-14](#) illustrates an OFD for a financial management system. From this diagram, it is apparent that the central subsystem relates to financial transactions, and all other subsystems are related to this. For additional illustrations, see [Figures 6-7](#), [Figure 6-11](#) and [Figure 9-7](#) also.



[Figure 6-14](#). OFD for a Financial Management System

The term object flow diagram is to some extent a matter of semantics: in an OO environment it may be used, or an ITC may render it unnecessary; in a traditional (FO) environment a DFD would have been used.

6.7 Other Contemporary Diagramming Techniques

Still, other diagramming techniques abound:

- The *entity-relationship diagram* (ERD) will be introduced in [chapter 10](#), but a full treatment is left for a course in database systems. The corresponding *object-relationship diagram* (ORD) of the OO paradigm will also be introduced in [chapter 10](#).
- The *fern diagram*, *event diagram* and other OO techniques will also be left for your course in OOSE or OOM. However, they are discussed in appendices 4 – 6.
- The UML (*Unified Modeling Language*) notation includes *object structure diagram* (OSD), *use-case diagram*, ORD, *activity diagram*, *collaboration diagram*, among other techniques. These will be introduced in [chapters 9, 10](#), and [12](#), with additional discussion in appendices 4 – 6. However, a full discussion is best done in a course in OOSE or OOM.
- The *Warnier-Orr diagram* will be discussed in [chapter 12](#).

The remainder of this section will briefly introduce the *state transition diagram* and the *finite state machine*, while leaving the excessive details for more advanced courses.

6.7.1 State Transition Diagram

A state transition diagram is used, primarily in OOSE, to represent the allowable state transitions associated with an object. [Figure 6-15](#) provides an illustration of a state transition diagram for an **Employee** object type. All instances of this object type will be subject to the transitions represented in the diagram.

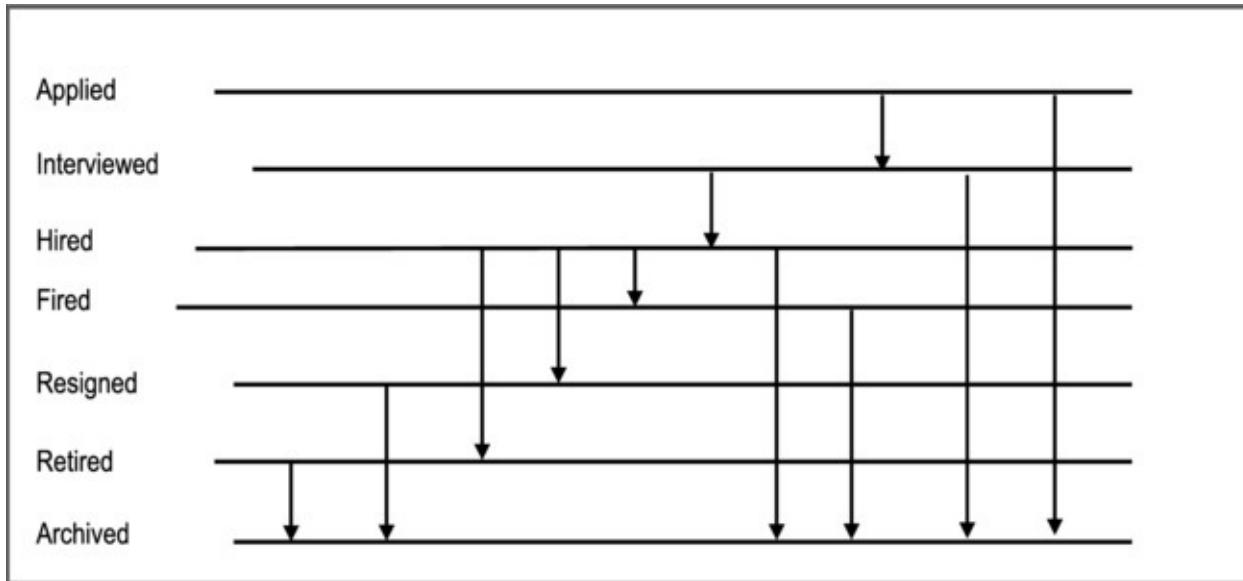


Figure 6-15. State Transition Diagram of Employee Object

6.7.2 Finite State Machine

An alternative to the state transition diagram is the finite state machine (FSM), also called the state diagram in some texts. Though existent before object technology (for instance in compiler design), FSMs find very useful application in here.

In an FSM (also referred to as state diagram) nodes are states; arcs are transitions labeled by event names (the label on a transition arc is the event name causing the transition). The state-name is written inside the node. [Figure 6-16](#) illustrates an FSM for the state transition diagram of [Figure 6-15](#).

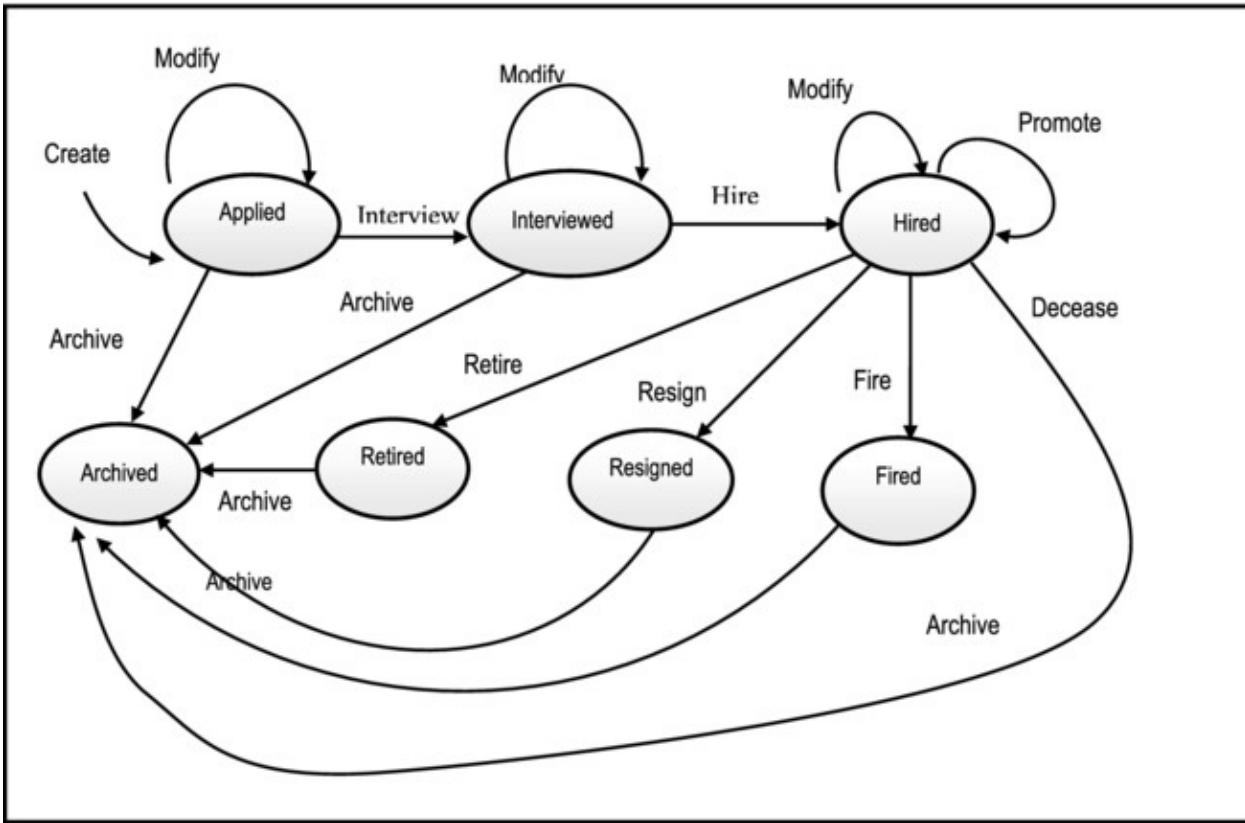


Figure 6-16. FSM for Employee

As you can see, a significant advantage of the FSM over the state transition diagram is that you can show operations that cause change of state as well as those that do not cause change of state.

6.8 Program Flowchart

The program flowchart is a traditional technique of the FO paradigm that represents functional logic. Its use in contemporary software design has been overtaken by other techniques such as algorithms (in pseudo-code), Warnier-Orr diagrams, and activity diagrams. It is assumed that from your earlier courses, you have gained mastery of algorithm development; as for Warnier-Orr diagrams, and activity diagrams, these will be covered later in the course ([chapter 12](#)). The symbols used for program flowcharting are shown in [Figure 6-17](#). The main structures (simple sequence, selection, and iteration) are also illustrated in the [Figure 6-18](#).

The main advantages of programming flow chart are easy debugging,

economy in writing and easy tractability. The main flaw is its limitation to non-parallel logic.

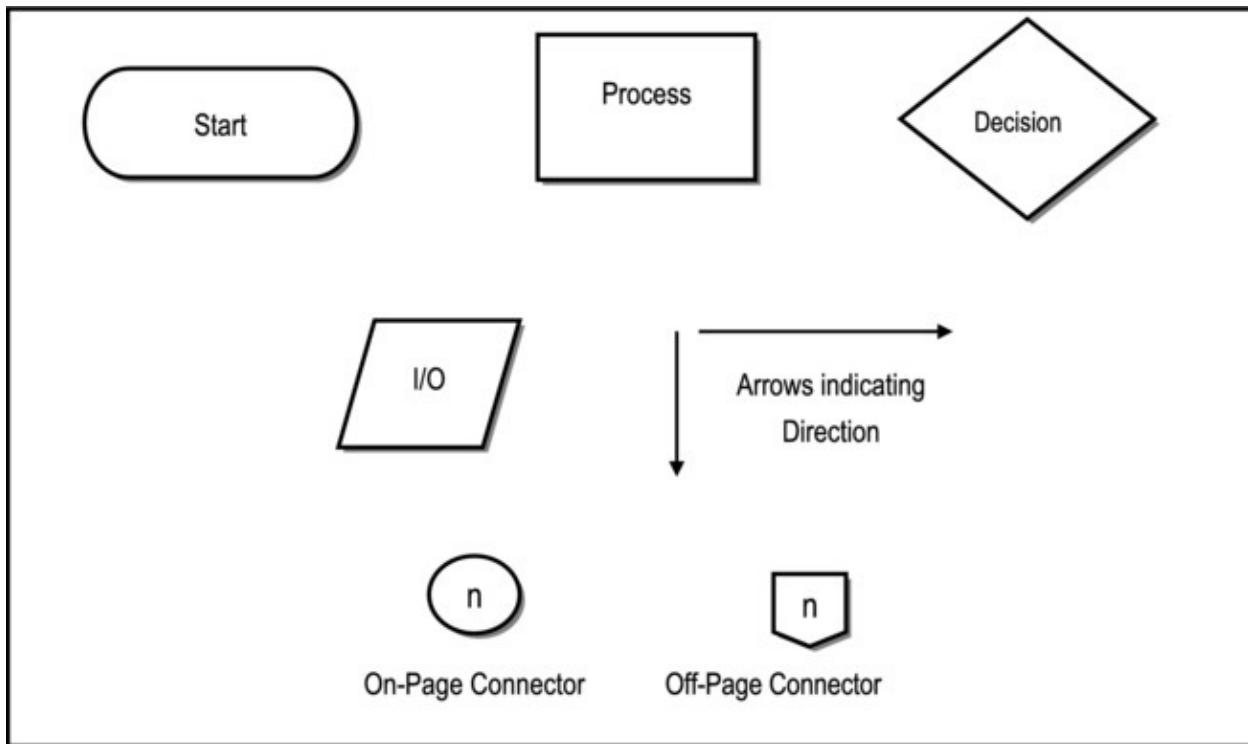
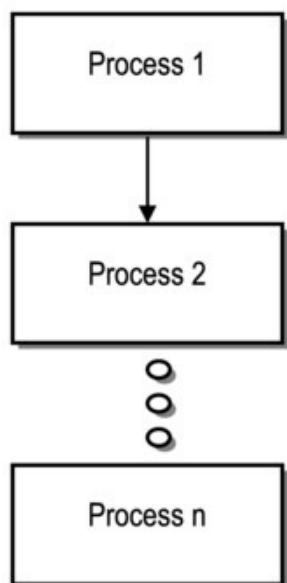
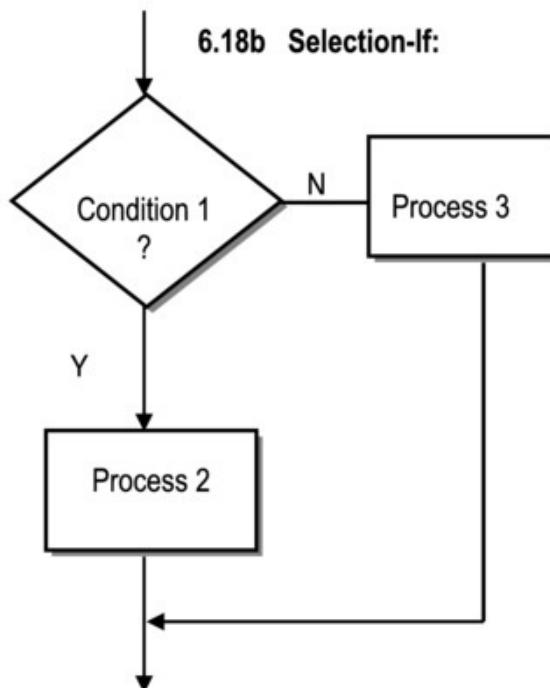


Figure 6-17. Symbols Used in Program Flow Chart

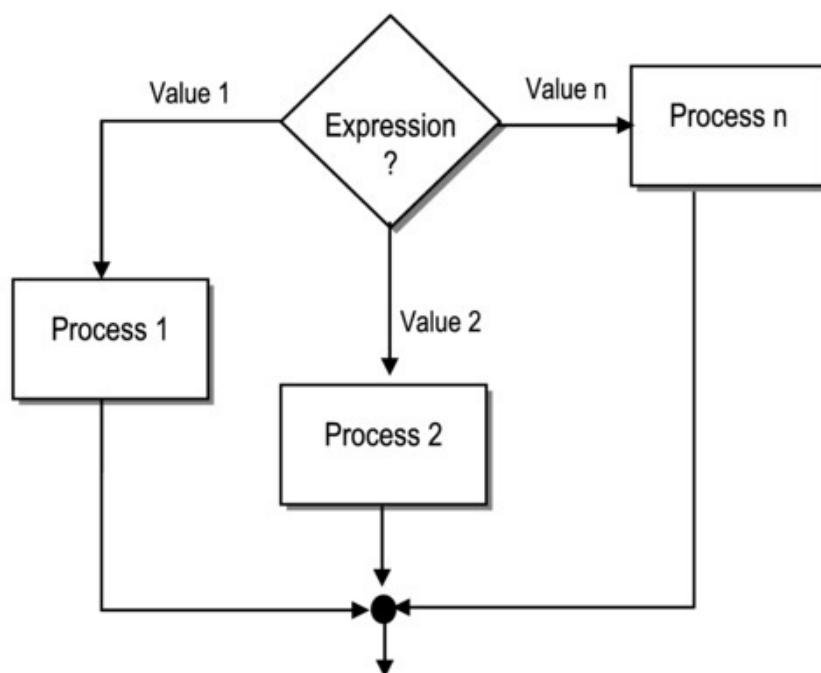
6.18a Sequence:



6.18b Selection-If:



6.18c Selection-Case:



6.18d Iteration:

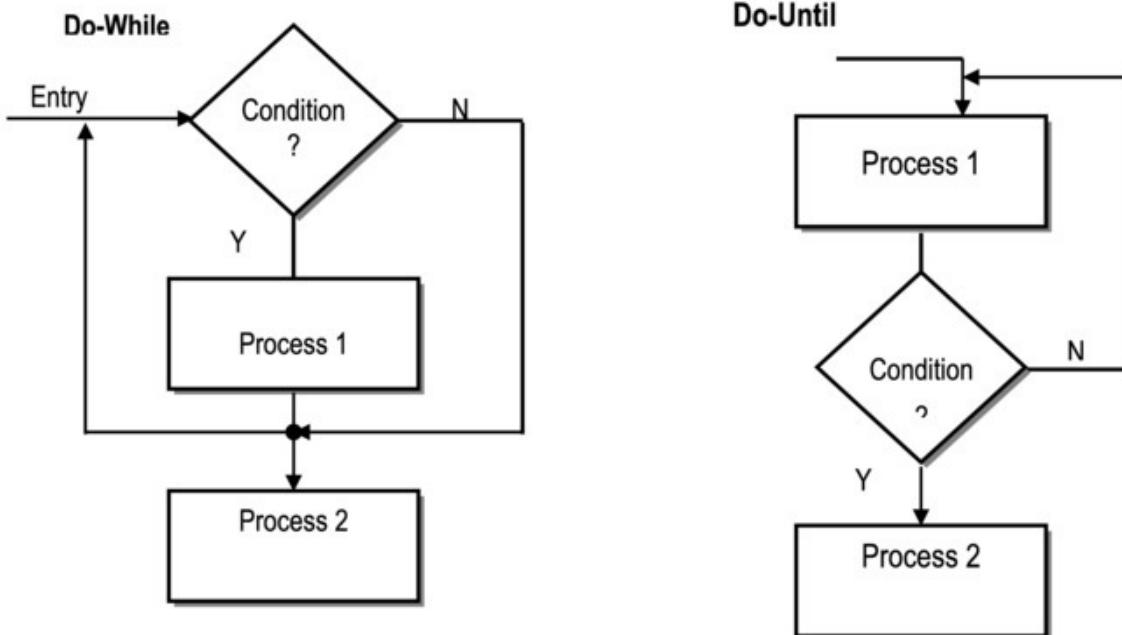


Figure 6-18. Control Structures for Program Flow Chart

6.9 Summary and Concluding Remarks

It is time once again to summarize what we have covered in this chapter:

- The software engineer relies on diagrams to document and communicate information concerning the requirements of a software system.
- Traditional system flow charts include information-oriented flowcharts, process oriented flow charts, HIPO charts, and data flow diagrams (DFDs).
- Traditional logic charts include decision tables, decision trees, and program flow charts.

- The procedure analysis chart (PAC) is useful in both FO and OO paradigms, particularly for business process reengineering (BPR).
- OO system diagrams include object flow diagrams (OFDs), O-R diagrams, state transition diagrams, finite state machine (FSMs), information topology charts (ITCs), and user interface topology charts (UITCs), fern diagrams, object structure diagrams, and UML diagrams.
- OO logic diagrams include event diagrams and activity diagrams.
- Diagrams that apply to both FO and OO paradigms include Warnier-Orr diagrams, E-R diagrams, finite state machine (FSMs), information topology charts (ITCs), and user interface topology charts (UITCs), procedure analysis charts (PACs), decision tables and decision trees.

There is only one way to gain mastery of these techniques — by applying them to problems. Take the time to review [appendix 8](#) and [appendix 9](#) once more; you will see some applications of some of the techniques. The next chapter will continue the discussion of some techniques not covered in this chapter.

6.10 Review Questions

1. What are the advantages of using diagrams?
2. What information is conveyed by the following diagrams?
 - Information oriented flowchart
 - Process oriented flowchart
 - Hierarchy input process output chart
 - Procedure analysis chart
 - Information topology chart
 - User interface topology chart

- Data flow diagram
- Object flow diagram
- State transition diagram
- Finite state machine
- Program flowchart

3. Compare the following diagrams:

- Information topology chart versus hierarchy input process output chart
- Process oriented flow chart versus data flow diagram
- State transition diagram versus finite state machine

4. A supermarket wishes to computerize its operation. The main issues to address are as follows:

- Purchases from external vendors are represented by invoices
- Purchases affect the Purchase Log (file), as well as Accounts Payable (file) and the Inventory (file)
- Inventory Management is affected by purchases and sales
- Every inventory item belongs to a category
- Sale of goods to customers (credit sales) as well as cash sales affects the Sales Log (file), the Accounts Receivable, as well as the Inventory

From the information given, propose the following:

- a. A POF or DFD of the system
- b. An ITC and UITC

5. A Library Management System (LMS) is being constructed. The software engineer is focusing on the information entity called **Book**. It was discovered that a **Book** object could be in any of the following states: **ordered, invoiced, loaned, shelved, archived, lost,**

stolen.

- Propose a finite state machine and state transition diagram for the Book object type.
- Compare the information conveyed by both diagrams.

6.11 References and/or Recommended Readings

[Foster, 1999] Foster, Elvis C. *Labour Market Information System: Thesis*. Mona, Jamaica: Department of Mathematics and Computer Science, University of the West Indies, 1999. See section 4.4.1.

[Harris, 1995] Harris, David. *Systems Analysis and Design: A Project Approach*. Fort Worth, TX: Dryden Press, 1995. See [chapter 4](#).

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [chapter 7](#).

[Long, 1989] Long, Larry. *Management Information Systems*. Eaglewood Cliffs, NJ: Prentice Hall, 1989. See [chapter 12](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. [Chapter 4](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [chapters 11 and 12](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See [chapter 8](#).

CHAPTER 7



Decision Models for System Logic

The previous chapter examined various system flow charts, which form part of the inventory of diagramming techniques employed by the software engineer. This chapter continues the discussion of diagrams by focusing on techniques used to represent and manage system logic. In the traditional FO paradigm, decisions are represented and analyzed by use of structured language (pseudo-code), decision tables or decision trees. In the more contemporary OO paradigm, the traditional methods are still applicable, but in addition to system rules. The main challenge is the identification of these decision issues; they are not always obvious. Here, skill and experience in information gathering are precious virtues.

The chapter will proceed with discussions in the following areas:

- Structured Language
- Decision Tables
- Decision Trees
- Which Technique to Use
- Decision Techniques versus Flowcharts
- System Rules
- Summary and Concluding Remarks

7.1 Structured Language

Decisions can be represented by use of structured English in pseudo-code-like manner. Use of the standard control structures for selection, iteration and

recursion is common (you should be familiar with these from your earlier programming courses). As usual, indentation improves readability. It is assumed that you know how to write a pseudo-code; therefore, nothing further will be said on this matter. Decisions may also be presented in a tabular manner as illustrated in [Figure 7-1](#).

If	Condition	Action
Student Balance:	Semester Tuition or more	Disallow registration
	Less than Semester Tuition	Allow registration for the difference
Student GPA:	Less than 2.0	Disallow registration
	2.0 or more	Allow registration
Course Prerequisite:	Done successfully	Allow registration
	Not done successfully	Disallow registration

[Figure 7-1.](#) Decisions to be Taken on Student Registration at a College or University

7.2 Decision Tables

A decision table is a tabular technique for describing logical rules. It serves as an aid to creative analysis and expresses a business situation in a cause-effect relationship. The decision table provides an excellent means of communication between users and software developers. It forces the software engineer to be objective in the decision making process, and to consider all the decision alternatives. [Figure 7-2](#) shows the basic components of a decision table.

Condition Stub	Condition Entries (Y/N)
Action Stub	Action Entries (Xs)

[Figure 7-2.](#) Structure of a Decision Table

7.2.1 Constructing the Decision Table

The following guidelines are useful in the construction of a decision table:

- Make small tables (maximum four conditions).

- All possible rules must be represented. A rule is simply a condition entry combined with an action entry.
- Every rule must have an action.
- Rules must be unique and independent.
- Define all alternate outcomes. In particular, note that N conditions $\Rightarrow 2^n$ outcomes (rules).
- Develop a set of conditions that yields each outcome.
- Assign a decision for each outcome.

Example 1: Consider as an example, an airline ticket seller is given the following guidelines:

1. There are two classes of tickets - first class and economy.
2. If a request is for first class and the space is available then reserve first class seats. If request is for economy and space available, reserve economy seat.

[Figure 7-3](#) shows a faulty decision table for the situation described. In the figure, the guideline that states that every rule must have an action is not met; the table is therefore incorrect. [Figure 7-4](#) shows a correct decision table. Notice that two additional conditions have been introduced.

Airline Booking				
If 1st Class	Y	Y	N	N
If Space Avail.	Y	N	Y	N
Reserve First Class	X			
Reserve Economy			X	

There are two problems with this decision table:

1. All possible conditions are not shown.
2. Every rule must have an action.

[Figure 7-3](#). Incorrect Decision Table for Airline Problem

	Rules															
Conditions/Actions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Request 1st class	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Req. Space Available	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Accept Alternative Class	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Alt. Space Available	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Reserve 1st class	X	X	X	X											X	
Reserve Economy					X				X	X	X	X				
Standby 1st class						X	X									
Standby Economy															X	X
Standby Either					X									X		

Figure 7-4. Correct Decision Table for Airline Problem

7.2.2 Analyzing and Refining the Decision Table

Two principles are used to analyze and refine the decision table: elimination of redundancies, and avoidance of ambiguities. Let us examine each.

Elimination of Redundancy: *Redundancy* occurs when two rules result in the same action and the condition entry responses are the same except for the last condition. The two rules can be combined (i.e. one is discarded). In Figure 7-4, rules 1&2, 3&4, 7&8, 9&10, 11&12, 15&16 constitute redundancies. For each pair, the second rule is eliminated; the revised table is shown in Figure 7-5.

Conditions/Actions	1	5	6	7	9	13	14	15
Request 1st class	Y	Y	Y	Y	N	N	N	N
Requested Class Available	Y	N	N	N	Y	N	N	N
Accept Alternative Class	-	Y	Y	N	-	Y	Y	N
Alternative Class Available	-	Y	N	-	-	Y	N	-
Reserve 1st class	X					X		
Reserve Economy		X			X			
Standby 1st class				X				
Standby Economy								X
Standby Either			X			X		

Note: Don't Care condition entries are indicated by '-'.

Figure 7-5. The Enhanced Decision Table for Airline Problem

Avoidance of Ambiguity: If two equivalent sets of conditions require different actions, the table is said to be *ambiguous*. Ambiguity occurs when *don't care* situations exist (indicated by dash (-)). [Figure 7-6](#) illustrates. Ambiguities are also referred to as contradictions. When they occur, you must revisit the analysis that led to them, and make appropriate adjustments.

Condition/Actions	1	2	3	4	5	6	7	8
Condition 1	Y	Y	Y	Y	N	N	N	N
Condition 2	Y	Y	N	N	Y	Y	N	N
Condition 3	-	N	-	N	Y	N	Y	N
Action 1	X			X	X			
Action 2			X			X		
Action 3		X					X	

Figure 7-6. Illustration of Ambiguity

7.2.3 Extended Decision Table

In situations where the number of conditions exceeds four and/or each condition can have more than two outcomes, the decision table becomes large and unwieldy. In these cases an extended decision table can be useful.

Extended tables are normally used to save space. [Figure 7-7](#) provides an example. Note that statement(s) made in the condition stub are incomplete so that both stub and entry must be combined in order to obtain the intended message.

Conditions/Actions	R1	R2
If Assets >=	10,000	8,000
Grant 5,000 loan	X	
Grant 4,000 loan		X

Figure 7-7. Example of Extended Entry Decision Table

Extended-entry tables also reduce the possibility of redundancy and contradiction as [Figure 7-8](#) illustrates. In this table, seven scenarios for action are represented, along with a default scenario, if none of the other seven scenarios is true. As an exercise, try to convert this to the most optimized system logic possible (you may represent the logic using pseudo-code, Pascal, C++ or Java). The exercise should also help you to see that the table can be further refined.

Conditions/Actions	R1	R2	R3	R4	R5	R6	R7	ELSE
Cost of The Item A: Cost < \$ 15 B: \$ 15 <= Cost <= \$ 60 C: Cost > \$ 60	-	A	A	B	B	C	C	
Order Quantity D: Qty. < 100 E: 100 <= Qty. <= 175 F: Qty. > 175	D	E	F	E	F	E	F	
Order Immediately		X		X		X		
Check Supervisor			X		X		X	X
Wait or Regular Order	X							

[Figure 7-8.](#) Extended Entry Decision Table for Inventory Ordering

7.3 Decision Trees

The decision tree is useful when complex branching occurs in a structured decision process, and it is essential to keep a string of decisions in a particular sequence. It therefore shows the relationship among decision factors. This is not shown by a basic decision table, and is poorly handled by a flowchart.

The decision tree used in software engineering and systems analysis is not as complicated as that used in management science, where probabilities and monetary expectations are shown. Essentially, the tree shows conditions and actions in a completely structured decision process.

Two symbols used are in the construction of a decision tree: an oval shape is used to symbolize **if condition**; a square shape symbolizes **then action**. [Figure 7-9](#) illustrates a decision tree for a sale operation.

Decision trees are preferred to decision tables when any of the following situations hold:

- The process(es) concerned is (are) accomplished in stages.
 - The logic is asymmetrical.
 - The decision conditions and actions are related.
-

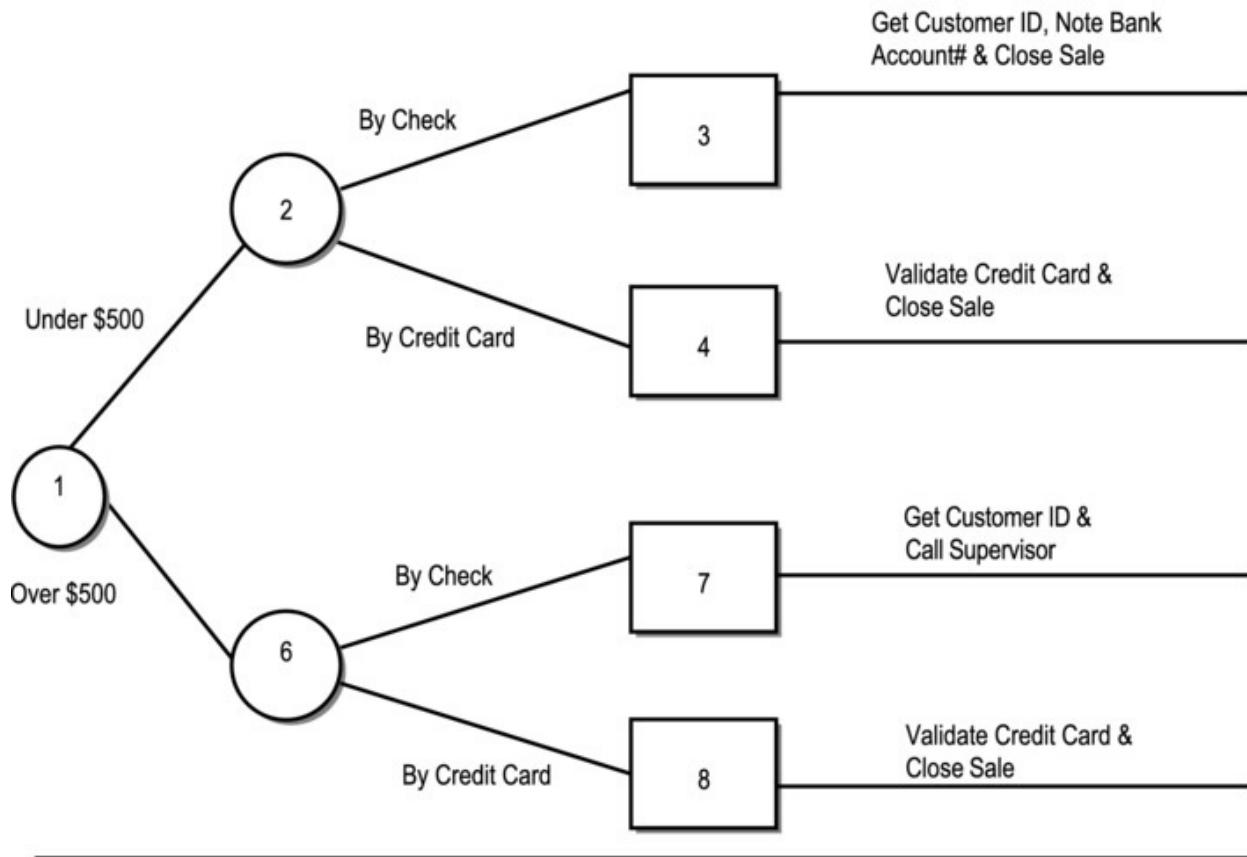


Figure 7-9. Decision Tree for a Sale Operation

Comparison of Decision Tree with Decision Table

How would you compare decision tables with decision trees? To get you started, here are some pointers:

- The decision tree emphasizes the sequential relation of decision conditions. This is impossible with the decision table.
- The decision tree handles quite effectively, situations where

certain conditions and actions apply, given specific circumstances, but do not apply, given other circumstances. This is more difficult to illustrate via the decision table.

- The decision tree is more readily understood by users than the decision table.
- Unlike the decision table, the decision tree effectively handles scenarios involving more than four conditions.
- The decision table is more effective than the decision tree at avoiding redundancies and ambiguities. It is also more concise than the decision tree.
- The decision table is more easily incorporated in a CASE tool than a decision tree.

7.4 Which Technique to Use

Use structured language when any of the following holds:

- There are many repetitious conditions.
- Communication to end-users is paramount.

Use decision tables when any of the following holds:

- Complex combinations of conditions, actions and rules exist.
- A method that effectively avoids impossible situations, redundancies and contradictions is required.
- There are no more than four condition entries.

Use extended decision tables when any of the following holds:

- There are more than four condition entries.
- It is desirous to save space by representing the decision scenarios in a concise manner.
- There are (or may be) conditions that have more than two possible outcomes.

- It is desirous to reduce/eliminate redundancies and ambiguities.

Use decision trees when any of the following holds:

- The sequence of conditions and actions is critical.
- Not every condition is relevant to every action (branches are different).
- There are more than four condition entries.

7.5 Decision Techniques versus Flowcharts

Following is a comparison between decision techniques as discussed in this chapter and flow charts as discussed in the previous chapter. The comparison is based on advantages and disadvantages of both sets of methodologies.

Flowcharts provide the following advantages:

- Flowcharts are useful in identifying bottlenecks, delay factors, redundancies, and other system flaws.
- They are effective in depicting the movement of data.
- They are also effective in depicting transitions between system states.
- They are useful in the representation of system logic.
- They are superb in emphasizing the interrelatedness of system components – subsystems, operations, data storage, user interfaces, etc.
- They are excellent for easy communication with end users.

The following disadvantages are associated with flowcharts (if a CASE tool that supports the diagrams is used, these disadvantages are minimized):

- Flowcharts can be difficult to draw, especially for large,

complex systems. Diagrams may be quite complex, particularly when there are several paths to consider.

- It may be difficult to determine whether the total problem is covered or whether superior alternate methods exist.
- Revision may be as difficult as drawing a complicated chart.

Decision techniques provide the following advantages:

- Structured format makes for easy drawing and clarity of presentation.
- Applications involving complex interactions of input variables are well documented and represented in a simple manner.
- Diagrams used are easy to revise and maintain with or without CASE.
- There is easy communication with end users.

The following disadvantages are associated with decision techniques:

- With decision techniques, there is no indication of the movement of data.
- They do not assist in identifying system bottlenecks, redundancies or delays as flowcharts.
- They do not represent system state transitions.
- They do not show the interrelatedness of the system components, only decision conditions.

Decision techniques and flow chart techniques are most effective when they are used to complement each other. The software engineer decides which techniques will be used to represent different aspects of the system. In an integrated CASE environment, these tools are all integrated as the system is modeled.

7.6 System Rules

[Figure 7-10](#) proposes a ten-step approach for the development of software using the OO paradigm. The upper portion of the diagram relates to the requirements specification (RS), and was introduced in [chapter 4](#) (Figure 4-1). The lower portion relates to the *design specification* (DS) and the actual development; we are yet to discuss these issues (discussion of the DS commences in [chapter 9](#)). As can be seen, the specification of system rules begins in the first portion of the schedule (item 5). These rules are subsequently refined and applied appropriately during the second portion of the schedule (item 8). Rules form an integral part of the system logic. The rest of this section focuses on essential details you should know about system rules.

Requirements Specification Activities

1. Refine your problem definition & proposed software solution.
2. Conduct your information gathering and identify:
 - Subsystem(s) and/or modules
 - Object types (information entities)
3. Identify or define operations for the various object types.
4. Provide a system overview which includes any convenient combination of the following:
 - System narrative
 - Information topology chart (ITC)
 - Object flow diagram (OFD)
5. For each subsystem, provide the following:
 - System overview (item 4)
 - Entity-Relationship (E-R) or Object-Relationship (O-R) diagram
 - Storage requirements
 - Operational requirements
 - System rules
 - Interface requirements (if applicable)
 - System constraints (if applicable)

Design Specification to Actual Development Activities

6. Refine object types (in terms of structure and operation). Note: If an OO-ICASE tool is used, each object type is defined in terms of name, attributes, and operations. They are automatically linked by the software system as defined in the O-R diagram. In the absence of an OO-ICASE tool, the linkages are not automatic. Alternately, prepare an O/ESG for each object type.
7. For each object type, define the following:
 - a state transition diagram, or a FSM (preferably FSM);
 - a collaboration diagram;
 - a set of activity diagrams.

Note:

- Your use of diagrams may be constrained by the CASE tool being employed.
- In the context of an OO-CASE tool, the diagrams are automatically linked; update of one ripples to the others.
- The state transition diagram (or FSM) should precede the activity diagram; the former aids the development of the latter.

8. Refine rules for each object type (class). Rules may also be refined and applied to operations, reflecting the business policy of the organization. Note: If you are using an OO-GUI superimposed on a RDBMS (for example: Delphi, Team Developer, etc.), it may be prudent to include as many of the business rules as possible in the relational database.
9. Develop operation specification for each operation.
10. Develop and test operations for system implementation.

Figure 7-10. The OO Software Construction Process

Before proceeding, we need to define two new terms in the OO paradigm:

- *Object Structure Analysis* (OSA) includes all analysis that relate to the structure of object types comprising the software system.
- *Object Behavior Analysis* (OBA) includes all analysis that relate to the behavior of system operations comprising the software system.

Martin's OO modeling pyramid (see [Martin, 1993]) identifies four levels of *information engineering* activities in the organization: enterprise modeling, business area analysis, system design, and system construction; it also shows how OSA and OBA are related to these activities. The term *information engineering* simply refers to a focused area of software engineering that relates to the efficient operation of business organizations (see appendices 3 and 4). Based on Martin's OO pyramid, [Figure 7-11](#) has been prepared. It also helps you to have a better appreciation of the schedule presented in [Figure 7-10](#).

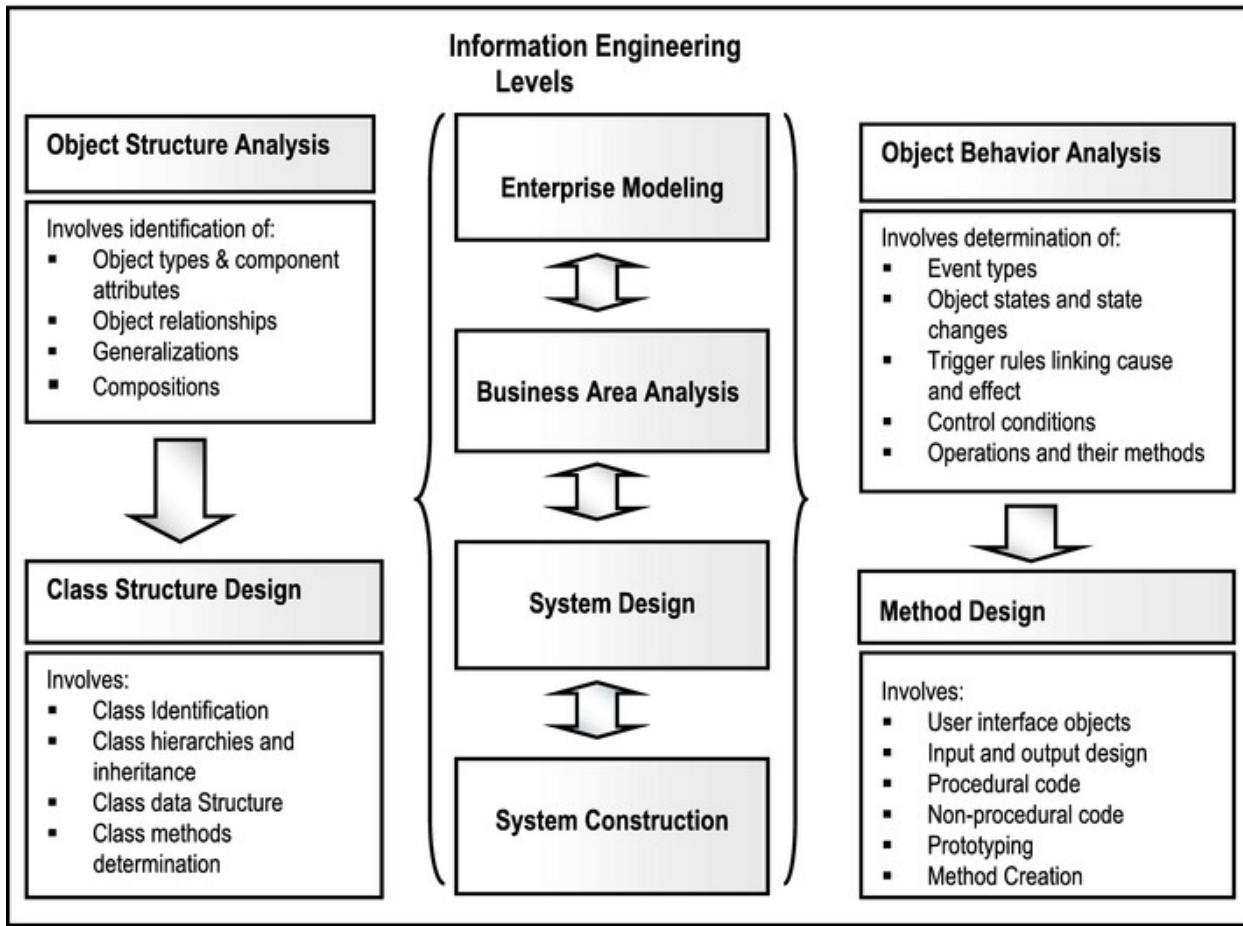


Figure 7-11. OO Modeling Pyramid

7.6.1 Rule Definition

A *rule* is a guideline for the successful and acceptable implementation of a process or set of operations. Failure to observe the rule could result in unacceptable system results. With traditional software engineering methodologies, rules are usually hidden in the code of system functions. In the OO paradigm (thanks to non-procedural languages), we want to define rules as an integral (encapsulated) part of the definition of (the operations of) a class. Code is then automatically generated from this.

Examples of Non-procedural Languages:

Examples of non-procedural languages are: FOCUS, RAMIS, NOMAD, NATURAL, IDEAL, MANTIS, Structured Query Language (SQL), and Knowledge Query Language (KQL).

Rules are particularly helpful in designing the desired behavior of the system

objects. They represent the laws about how the business is run. Rules may be in any of the following broad categories:

- Relating to object structure analysis (OSA) and inter-object relationships (business rules).
- Relating to object behavior analysis (OBA) and business policies of the organizations.

Rules must be rigorous, precise, concise, clear, and easily understood by end-users.

Example 2: The following is an example of an OBA rule:

When a student's GPA is than 3.6 or greater, he/she goes on the Dean's list.

7.6.2 Declarative versus Procedural Statements

Conventional programming languages are procedural. Declarative languages declare facts and rules, without specifying processing procedure. Declarative statements are more concise and easier to understand and validate than procedural statements.

Example 3: Suppose that we are storing information about students in a database file called **Student**. Assume further that the property **StudGPA** stores the GPA of the student. The following SQL statement produces a logical view (called **DeanList**) of all students with a GPA of at least 3.6:

```
Create View DeanList AS SELECT * FROM Student WHERE StudGPA >= 3.6;
```

Because declarative statements are more flexible, it is preferable to build systems from declarative languages linked to OO-CASE tools, rather than procedural languages.

With an OO-CASE tool, rules are constructed with the use of a rule editor. Code is automatically generated so the rule can be executed immediately. Two examples of such products are LiveModel (previously called are Object Management Workbench) from Intellicorp and Rational Rose from Rational Software (now a division of IBM).

7.6.3 Types of Rules

Lee's [Lee, 2002] refinement of Martin's [Martin, 1993] categories of system rules include eight (8) categories as described below:

Data Integrity Rule: A rule of this type states that condition about an attribute must be true (for example, marital status must be **married**, **single**, **widowed**, **separated** or **divorced**).

Relationship Integrity Rule: Such a rule states that something about a relationship between two object types must be true (for example, a department cannot have more than 25 employees).

Derivation Rule: A derivation rule prescribes a relationship between a dependent variable and a set of independent variables; it typically relates to calculated values (for example **InvoiceAmount** = Summation of **Price * Quantity** for each line item on the invoice).

Pre-condition Rule: This type of rule states that a condition must be true before an operation is executed (for example, an employee object cannot be fired unless it is in a state of **Hired**).

Post-condition Rule: This type of rule states the condition(s) which must exist after an operation is performed (for example, after the **FireEmployee** operation is executed, the stat of the employee object is **Fired**).

Action Trigger Rule: A rule of this type defines the causal relationship between an event and the Operation that triggers it (for example, when the result of an interview is recorded, the appropriate action, commensurate with the recommendation, must be taken).

Data Trigger Rule: This type of rule defines the causal relationship between an attribute's status and an action (for example, when a student's GPA falls below 2.0, send an alert letter).

Control Condition Rule: Such a rule handles the situation in which an operation is caused by multiple triggers. The control condition rule determines what combination of circumstances will cause the operation to execute (e.g., if the student's GPA is below 2.0 AND the student has registered for more than 12 credits, him/her student on probation).

Traditionally, data integrity rules, relationship integrity rules, and derivation rules are described as *business rules*. The term *stimulus/response rules* is sometimes used to refer to action trigger rules, data trigger rules, and control condition rules.

[Figure 7-12](#) provides the recommended constructs for the different types of rules. [Figure 7-13](#) provides some examples.

1. For stimulus/response rules:
IF <Condition> THEN <Action> // Preferred for data triggers
Or
WHEN <Event> IF <Condition> THEN <Action> // Preferred for action triggers & control conditions
2. For business rules:
IT MUST ALWAYS BE THAT <Statement of Fact>
3. For pre-condition rules:
BEFORE <Operation> IT MUST BE THAT <Statement of Fact>
4. For post-condition rules:
AFTER <Operation> IT MUST BE THAT <Statement of Fact>
5. For derivation rules:
WHEN <Event or Condition> THEN <Statement of Fact>

[Figure 7-12. Recommended Constructs for Different Types of Rules](#)

IT MUST ALWAYS HOLD THAT

An employee belongs to one and only one department.

IT MUST ALWAYS HOLD THAT

The number of employees who are managers with salary greater than \$2.5 M is less than 5.

IT MUST ALWAYS HOLD THAT

Sex is either 'M'ale or 'F'emale.

IF Student.GPA \geq 3.90

THEN classify student as "Summa Cum Laude"

BEFORE Bruce.Interview() **IT MUST BE THAT** (Bruce.GetStatus() = "Applied")

Figure 7-13. Examples of Rules

7.7 Summary and Concluding Remarks

Let us summarize what has been discussed in this chapter:

- The traditional methods of representing decisions are via structured language, decision tables and decision trees.
- The structured language may be in the form of pseudo-code or a tabular representation. It is applicable in simple situations where communication to end-users is of paramount importance.
- The decision table has four main sections: the condition stub, the condition entries, the action stub and the action entries. The decision table must be free of redundancies and ambiguities. Decision tables are useful in situations where there are complex combinations of four or less conditions, and it is desirable to efficiently represent all the possibilities while avoiding ambiguities as well as redundancies.
- The extended decision table is a modification of the basic decision table in order to provide more flexibility. It is particularly useful in any combination of the following situations: more than four conditions exist; not all the

conditional possibilities are relevant; and it is desirable to be concise but accurate.

- The decision tree is a hierarchical graphical representation of the decision problem. It is useful in any combination of the following situations: the sequence of conditions and actions is critical; not every condition is relevant to every action (i.e. the branches are different); there are more than four condition entries.
- A system rule is a guideline for the successful and acceptable implementation of a process or set of processes.
- There are eight types of rules: data integrity rules, relationship integrity rules, derivation rules, pre-condition rules, post-condition rules, action trigger rules, data trigger rules and control condition rules.
- Rules must be clearly specified according to predetermined formats.

Traditionally, expert systems were based on an important component called the *inference engine*.

An inference engine is a collection of facts and rules about a specific area of knowledge and facilitates deductions based on established techniques of logical reasoning. Special software was used for recording of such facts.

Object technology facilitates the easy, more pragmatic construction of inference engines for expert systems. It's also more exciting.

Some rules are to be visible to end users, so that they may check them in a workshop. These include business policy rules, derivation rules. There are other rules that are best kept invisible (transparent) to the end user, to avoid confusion. These include rules for technical design, integrity rules, rules internal to the software (OO-CASE tool and/or Repository) being used.

Mastery of the techniques discussed since [chapter 3](#) places the software engineer in an excellent position to prepare an impressive RS. This time, your RS should have a refined project schedule. The next chapter will provide guidelines on how to achieve this. Before you proceed, take a second look at the sample requirements specification included in [appendix 9](#); it should provide you with useful insights.

7.8 Review Questions

1. What are the decision techniques that have been discussed in this chapter? What circumstance(s) would warrant the use of each technique?
2. Consider the registration process at your institution, or a similarly complex process at an organization that you are familiar with:
 - Identify the different circumstances (conditions) that must be considered.
 - Determine an appropriate action for each circumstance.
 - Represent your findings using one of the decision techniques discussed in the chapter.
 - Develop a software application to address the problem, using Delphi, Visual C++, C++ Builder, or any other RAD tool that you are familiar with.
3. What is a system rule? Briefly describe the different types of rules discussed.
4. Still considering the registration process at your institution, answer the following questions:
 - a. Assume that there is an operation called **Student.RegisterCourse**. What pre-condition might exist for registering for a particular course?
 - b. Assume that there is an operation called **Student.FindGPA** that calculates the GPA of a student from an array field **Student.Grades** which contains the student's grades. Propose a derivation rule for the GPA.

7.9 References and/or Recommended Readings

[Jacobson, 1992] Jacobson, Ivar. *Object Oriented Software Engineering: A Use Case Approach*. Boston, MA: Addison-Wesley, 1992.

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [chapter 9](#).

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development With UML and Java*. Upper Saddle River, NJ: Prentice Hall, 2002. See [chapters 9](#).

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1993. See chapter10.

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See sections 4.4 and 14.3.

CHAPTER 8



Project Management Aids

In this chapter, we examine three project management aids — PERT/CPM, Gantt charts, and project management software. The techniques are useful for managing resources, as well as monitoring of targets and expenditure during a software engineering project. The techniques are also useful in planning the software engineering project — a process which commences with the initial software requirement (ISR), is refined in the requirements specification, and further refined in the design specification. The chapter proceeds under the following captions:

- PERT and CPM
- The Gantt Chart
- Project Management Software
- Summary and Concluding Remarks

8.1 PERT and CPM

PERT is an acronym for *Program Evaluation and Review Technique*; CPM is an acronym for *Critical Path Method*. Developed in the 1950's by the US Navy, the two techniques are normally used together; they constitute the most popular methodology for managing projects.

Projects managed with PERT/CPM need not be related to information systems or software engineering; the technique is applicable to all disciplines.

Among the advantages of PERT/CPM are the following:

- The project is represented graphically, showing important

events in some form of chronology.

- The critical path is identified (no delay is allowed along the critical path).
- The technique (CPM) allows for analysis and management of resource scheduling.
- The technique shows areas where tradeoffs in time or resources might increase the possibility of meeting major schedule targets.

In planning and preparing the PERT/CPM model for a project, the following steps are required:

1. Itemize activities in a tabular form showing for each activity its immediate predecessor, description, estimated time (duration).
2. Draw PERT Diagram using either the *activity-on-arrow* (AOA) approach or the *activity-on-node* (AON) approach.
3. For each activity (event), calculate the earliest start time (ES), the earliest finish time (EF), the latest start time (LS), and the latest finish time (LF). Also indicate the activity's duration (D).
4. Determine the critical path — the path through the network that has activities that cannot be delayed without delaying the entire project.
5. Conduct a sensitivity analysis to aid and inform the resource management process.

These steps are best illustrated by an example, so we will proceed with one.

Example 1: Let us suppose that we are involved in a software engineering project to design, construct, and implement a software product. We will construct an activity table and then refine it into a PERT diagram.

8.1.1 Step 1: Tabulate the Project Activities

[Figure 8-1](#) shows the schedule for our software engineering project. Notice that

except for the starting activity, each activity has a predecessor. Each activity also has an estimated duration (typically expressed in days, but any unit of time may be used).

Activity	Predecessor	Description	Estimated Time (days)
A	--	Design System Architecture	30
B	A	Design Operation Specifications	12
C	B	Design Control Operations	8
D	B	Design Modification Operations	15
E	B	Design Inquiry/Report Operations	7
F	C	Code Control Operations	2
G	C	Prepare System. User Guide	5
H	F	Test Control Operations	2
I	D	Code Modification Operations	6
J	I	Test Modification Operations	4
K	H, J	Test Control/Modification Operations	2
L	E	Code Report Operations	3
M	L	Test Inquiry/Report Operations	1
N	K, M	Integration Test	4

Figure 8-1. Schedule of Activities for a Project

8.1.2 Step 2: Draw the PERT Diagram

The PERT Diagram is shown in [Figure 8-2](#). Note that the AON convention is followed. Note also the use of *dummy nodes* (activities) to improve the clarity and readability of the diagram.

The AON Convention is used. Each activity is represented in the form shown, where A represents the activity, and D its duration.

ES	A	EF
LS	D	LF

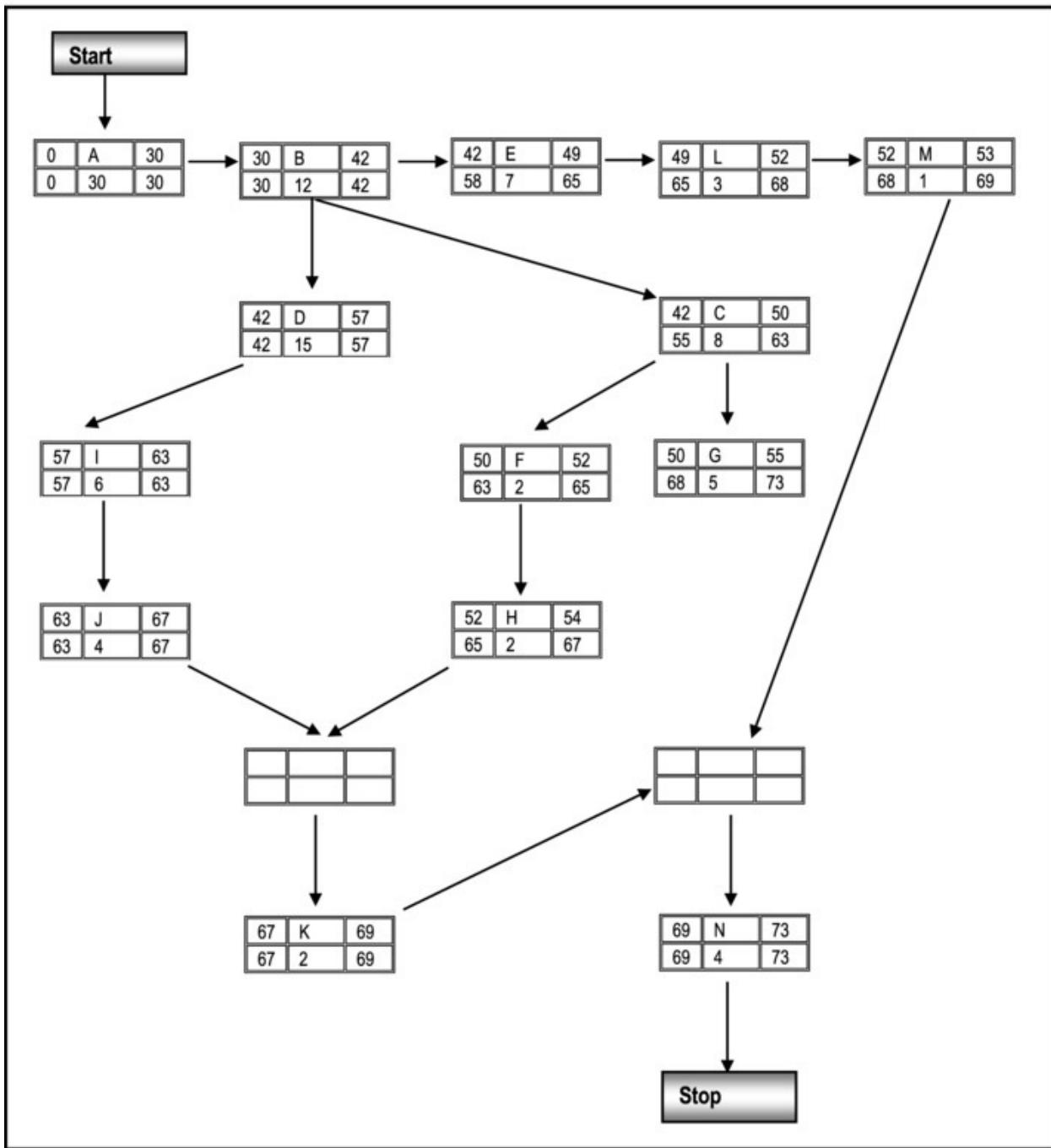


Figure 8-2. PERT Diagram for the Project

In keeping with convention, a start node and a stop node have been introduced. They are dummy nodes that have no duration.

8.1.3 Step 3: Determine ES, EF, LS and LF for each Activity

The ES, EF, LS, LF values for each activity is shown in [Figure 8-2](#). These values are to be calculated base on the formulae and rules shown in [Figure 8-3](#):

ES (Activity) = EF (Immediately Previous Activity) /*if only one activity immediately precedes*/
= Max [EF of all activities immediately preceding]

EF (Activity) = ES (Activity) + Duration of Activity

LS (Activity) = Min [LS of immediately succeeding activities] - Duration of Activity
= LF (Activity) - Duration of Activity

LF (Activity) = LS (Activity) + Duration
= Min [LS of all activities immediately following]

Slack (Activity) = LS (Activity) - ES (Activity)

Note:

1. ES of first activity is 0; LF of last activity, or terminal activities is equal to the earliest finish of the last activity.
2. ES and EF calculation originate at start and proceed to the end. LS and LF calculations originate at the finish and proceed backwards to the start.
3. A terminal activity is an activity that no other activity follows. In our example, G,N are terminal activities. For such activities, LF = EF of the last activity.

[Figure 8-3](#). Basic PERT Diagram Calculations

8.1.4 Step 4: Determine the Critical Path

The critical path is the path with zero slack. No delay can be allowed on the critical path. In the example, the critical path is **A B D I J K N**. The project should not be allowed to overrun the critical path time.

8.1.5 Step 5: Conduct a Sensitivity Analysis

PERT sensitivity analysis relates to two main matters: determination of the estimated time for each activity, and analysis of the prospect of *crashing* the project.

Determination of estimated time is usually done using the formulae shown in [Figure 8-4](#). It must be stated however, that in many cases, rather than calculating the estimated time on each activity, an experienced software engineer will place an estimated time period on each activity, based on the rigor of that activity, the talents and skills of the software engineering team, and comparative knowledge of the duration of similar activities on other project(s).

$$t_e = 1/6 (t_p + 4t_m + t_o)$$

$$SD = (t_p - t_o)/6$$

Where

t_e = estimated time;

t_o = optimistic time;

t_p = pessimistic time;

t_m = most likely time.

SD = standard deviation.

[Figure 8-4](#). Calculating Estimated Time for an Activity

Crashing is the process of shortening the project at increased cost. The cost of crashing is evaluated and weighed against the cost of missing the project deadline. A final decision is then made.

Note:

- Only activities on the critical path must be crashed.
- Crashing may be considered when the cost of crashing is less than the overhead cost for the *crash period* contemplated. The total cost of the project would then be reduced.

$$\text{Total Cost} = \text{Overhead Cost} + \text{Crashing Cost}$$

The critical path can be also used as a guide in resource planning and management, as well as cost management.

Example 2: Let us use the project of Example 1 to carry out a basic sensitivity analysis.

Suppose that the project cost is \$ 3,000 per week for two professionals, and that management can spend an additional \$ 1,000 to have the project reduced by one week. Would crashing be feasible?

Projected duration of project is 73 days i.e. 14.6 weeks, to be reduced to 13.6 weeks.

Original Project Cost = $14.6 * \$ 3,000 = \$ 43,800$

Limit management can spend = $\$ 43,800 + \$ 1,000 = \$ 44,800$

Payment rate is \$ 1,500 per week, per professional

Crashing by one week would attract one additional professional i.e. \$ 1,500, making

Total Project Cost = $13.6 * \$ 3,000 + \$ 1,500 = \$ 42,300$.

Since this is less than the maximum the company can spend, crashing is feasible.

Note: If (Crash Cost + Overhead Cost > Max. Management Can Spend), then crashing is not feasible.

8.2 The Gantt Chart

The Gantt chart was devised by Henry Gantt (1920's). The project is represented on a bar chart based on the following guidelines:

- Tasks are listed vertically.
- A horizontal time scale is used to indicate time duration.
- A bar is used to depict activities.
- Some indication is given of the percentage completion of each activity (for e.g. by color code).

With the advance of PERT / CPM, the Gantt chart is best used to emphasize the critical path activities. The main advantages are:

- The chart is simple and easy to read.
- The chart can be used to manage the progress of activities on a software engineering project.

[Figure 8-5](#) illustrates the Gantt chart for the project discussed in the previous section. On the diagram, the critical path activities are represented by grey bars, while non-critical-path activities are represented by bars that are not shaded. A third color code is often used to indicate the level of completion of certain activities.

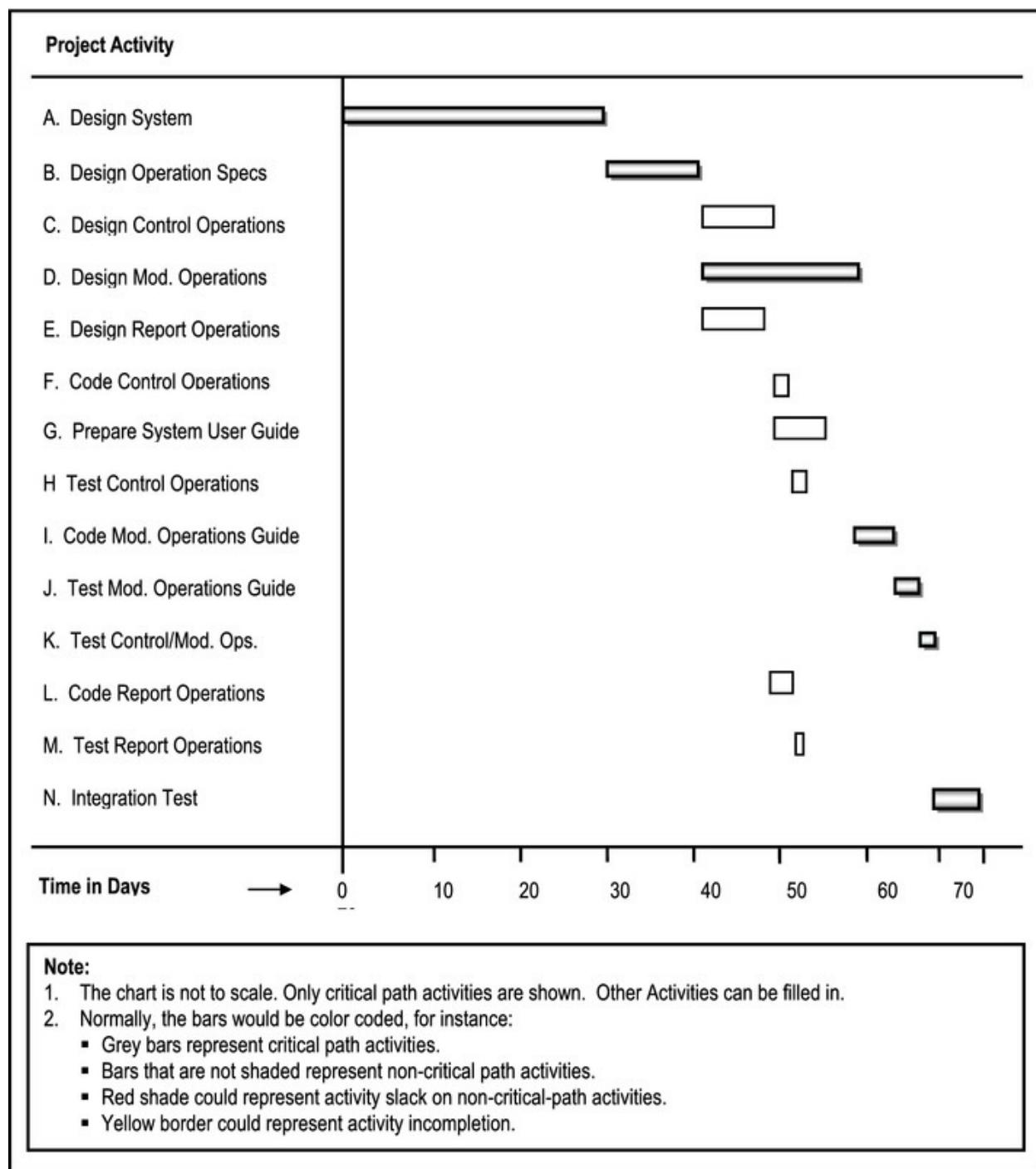


Figure 8-5. Gantt Chart for the Sample Project Showing Critical Path Activities

The Gant chart, combined with the PERT diagram constitutes a powerful project management tool. Following are some ways the two techniques may be employed:

- The lead software engineer can make informed decisions

about assigning certain responsibilities to members of the software engineering team. Since critical path activities cannot be delayed, such activities are usually assigned to stronger members of the team. Conversely, non-critical activities with longer slack times can be assigned to weaker members of the team.

- The project manager may use the techniques to assist in monitoring the progress of the project on a day-to-day basis.
- Should this become necessary, informed decisions about crashing can be made by identifying the critical path activities.

8.3 Project Management Software

You might be wondering, am I required to draw these PERT diagrams and Gant charts from scratch? The honest answer to this question is yes, you should know how to draw these diagrams from scratch. However, I believe what you want to know is whether there are software systems that can assist you in using these techniques. The answer to this inquiry is also a resounding yes. There is a wide range of project management software systems that are available in the marketplace. Reference [Capterra, 2008] provides a comprehensive list of several such software alternatives that you can peruse.

In choosing a project management software system, here are some desirable features that should be facilitated:

- Maintenance of a calendar of important dates
- Management of multiple projects
- Specification of project activities for each project
- Choice from techniques such as PERT diagram, Gantt chart, and any other technique available
- Automated generation of the desired diagram
- Ability to print diagrams or save them in different formats
- Management of other resources such as budget, as well as physical resources

- Management of project requirements
- Project estimations
- Scheduling of activities for team members
- Progress tracking for overall project
- Progress tracking for team members
- Cost-to-completion tracking for each project
- Risk assessment and management
- Ability to probe what-if scenarios

In addition to project management software products, some of the more sophisticated software planning and development tools that are available (as discussed in section 2.4.5) have project management facilities incorporated in them. Thus, the software engineer does not have to do all the hard work from scratch; you can use technology to make an otherwise very challenging job exciting and enjoyable.

8.4 Summary and Concluding Remarks

Here is a summary of what has been discussed in this chapter:

- A PERT diagram is a graphical representation of a project schedule that facilitates easy management of the project.
- Each node on the PERT diagram must have its name, duration (D), earliest start time (ES), earliest finish time (EF), latest start time (LS), and latest finish time (LF) clearly indicated.
- ES and EF calculations originate at the start point and proceed to the final point of the project. LS and LF calculations originate at the final point and work backwards to the starting point.

- The critical path through a PERT diagram is the path that has zero slack on all its activities.
- Crashing is the process of shortening a project at increased cost. If the sum of the cost of crashing and the overhead cost is within the limit that the organization is prepared to spend, then crashing is feasible.
- The Gantt chart is best used to emphasize the critical path activities.
- A project management software is a software system that facilitates easy management of a project. These products are readily available in the marketplace.

Armed with this knowledge, you can now incorporate an impressive project schedule in your RS, and thus complete your second major deliverable. The next section of the course discusses software design issues.

8.5 Review Questions

1. Clearly outline the steps involved in conducting a PERT-CPM analysis.
2. Explain how a PERT diagram and a Gantt chart may be used to assist in the management of a software engineering project.
3. A project involving the installation of a computer system consists of eight activities as shown in the activity table below:

<u>Activity</u>	<u>Immediate Predecessor</u>	<u>Time (weeks)</u>
A Equipment acquisition and Preparation	--	3
B Data conversion and Testing	--	6
C User Training	A	2
D Parallel Run	B, C	5
E Troubleshooting & Observation	D	4
F Fine-tuning	E	3
G Advanced User Training	B, C	9
H Signoff	F, G	3

- a. Draw the PERT network for this project.

- b. Identify the critical path.
 - c. What is the expected completion time for the project?
 - d. Identify three activities that can be delayed and for each, determine the slack.
4. LMX Software considering developing a new software product. The project activities identified so far are shown below:

<u>Activity</u>	<u>Immediate Predecessor</u>	<u>Time (weeks)</u>
A Prepare Requirements Spec	--	06
B Prepare Design Spec	--	08
C Prepare Development Team	A, B	12
D Develop and Test Module 1	C	04
E Develop and Test Module 2	C	06
F Develop and Test Module 3	D, E	15
G Conduct Comprehensive Sys Test	E	12
H Refine and Hand Over System	F, G	08

- a. Develop a PERT network for the project
- b. Identify the critical path.
- c. Determine the total project duration.
- d. Identify three activities that can be delayed and for each, determine the slack.

8.6 References and/or Recommended Readings

[Anderson, 1994] Anderson, David, Denis Sweeney and Thomas Williams. *Introduction to Management Science: Quantitative Approaches to Decision Making* 7th ed. New York, NY: West Publishing, 1994. See [chapter 10](#).

[Capterra, 2008] Project Management Software Directory.
<http://www.capterra.com/project-management-solutions>
 (accessed June 2008).

[Harris, 1995] Harris, David. Systems Analysis and Design: A Project Approach. Forth Worth, TX: Dryden Press, 1995. See [chapters 5 and 6](#).

[Peters, 2000] Peters, James F. and Witold Pedrycz. Software Engineering: An Engineering Approach. New York, NY: John Wiley & Sons, 2000. See [chapter 4](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. Software Engineering Theory and Practice 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapter 3](#).

[Sommerville, 2006] Sommerville, Ian. Software Engineering 8th ed Reading, MA: Addison Wesley, 2006. See [chapter 5](#).

[Taha, 1987] Taha, Hamdy A. Operations Research: An Introduction 4th ed. New York, NY: McMillan, 1987. See [chapter 13](#).

PART C



Software Design

The next five chapters will focus on the Design Phase of the SDLC. The objectives of this phase are as follows:

- To use the findings of the requirements specification to construct a model of the software that will serve as the blueprint to the actual development of the software product
- To thoroughly document this model

The activities of this phase culminate into the third major deliverable of a software engineering project — the design specification. Chapters to be covered include the following:

- [Chapter 9](#) — Overview of Software Design
- [Chapter 10](#) — Database Design
- [Chapter 11](#) — User Interface Design
- [Chapter 12](#) — Operations Design
- [Chapter 13](#) — Other Design Considerations

[REDACTED]

CHAPTER 9



Overview of Software Design

This chapter provides you with an overview of the software design process. It is the first on your experience towards the preparation of the next major deliverable in a software engineering project — the design specification. The chapter proceeds under the following captions:

- The Software Design Process
- Design Strategies
- Architectural Design
- Interface Design
- Software Design and Development Standards
- The Design Specification
- Summary and Concluding Remarks

9.1 The Software Design Process

Software design is a creative process that is perfected with experience. It cannot be fully learnt from a reading a series of lectures or chapters of a book; however, guidelines can be given. The objective of the software design process is the construction of a model that is representative of the required software system. The model must be accurate and comprehensive; it must conform to established software design and development standards; it must also meet the quality factors discussed in [chapters 1 and 3](#).

The design process starts with an informal design outline and undergoes

several stages of refinement until a final model is obtained. The key deliverable from the design phase is the *design specification* (DS). Much feedback (backtracking) occurs among the stages of the design phase, as well as between the preparation of the requirements specification and the preparation of the design specification. In fact, it is a good idea to work on both deliverables concurrently, since changes in either will trigger adjustments in the other.

[Figure 9-1](#) shows activities that go on during the design phase and what they lead to. Bear in mind that typically, some of these activities go on in parallel, rather than sequentially, so that feedback is constantly obtained and utilized. Below, these activities are clarified, but fuller discussion (on each activity) will follow in this and the next four chapters.

Design Activity	Resultant Design Spec Component
Architectural Design	System Architecture Specification
Interface Design	System Interface Specification
Database (Object Structure) Design	Database (Object Structure) Specification
Operations Design	Operations Specification
User Interface Design	User Interface Specification
Documentation Design	System Documentation Specification
Message Design	Message Specification
Security Design	Security Specification

[Figure 9-1](#). Important Software Design Activities

Architectural Design: Architectural design relates to the subsystems and/or modules making up the system, as well as their interrelationships. HIPO charts and system topology charts are useful diagramming tools that are used to represent the software model.

Interface Design: For each subsystem (module), its interface with other modules is designed and documented. The spin-off of interface design is the *system interface specification*, which must be unambiguous.

Object Structure Design: Object structure design relates to the data structures used in the system — object types (information entities), relationships, integrity constraints, data dictionary etc. Depending on the software engineering paradigm employed, it may be referred to as database design (in the case of traditional or hybrid approach), or object structure design (in the case of purely object oriented approach). Object structure design results in the *object structure specification* of the system. This will be further discussed in [chapter 10](#).

Operations Design: This involves preparing operation specifications for all the operational components of the software. Operations design also involves

categorization of the operations as well as design of algorithms used in the operations. It will be further elaborated in [chapter 12](#).

User Interface Design: User interface design relates to screen design, menu structure(s), input and output design and in some systems, a user interface language. This will be further discussed in [chapter 11](#).

Documentation Design: Documentation is an important part of software engineering. It includes all forms of product documentation (help system, users' guide, system manuals, etc.) and will be elucidated in [chapter 13](#).

Message Design: One important method of software communication with end users is via (error and status) messages. This aspect of software planning and construction is often ignored or belittled, to the horror of those who subsequently have to maintain the software. As you will see later ([chapters 12 and 13](#)), message design is a very important component of operations design.

Security Design: This relates to the various authority constraints that different users of the software product will have. How this is designed and the kind of security mechanisms that may be required will depend to a large extent on the type of software and its intended users. These issues will be addressed in [chapter 13](#).

The importance of software design as a precursor to software construction cannot be over emphasized. Success in the former often leads to success in the latter. Additionally, flawed design inevitably leads to flawed development. Note however, that in both cases, the implication is not necessarily reversible: You can have a flawed construction after a good design. Here are three important rules worth remembering (the first was learned from a former professor, E. K. Mugisa; the other two were constructed out of experience).

Rule 1: The sooner you run to the computer, the longer you stay there.

Rule 2: If it does not work on paper, it simply does not work.

Rule 3: Keep your design simple but not simplistic.

9.2 Design Strategies

The design approach may be *top-down* or *bottom-up*. Top-down design is traditionally associated with *function-oriented design* (FOD) and bottom-up with *object oriented design* (OOD). This is somewhat misleading, however, since it is possible to have top-down design that is object oriented (this is sometimes recommended), and bottom-up design that is function oriented.

Until the early 1990s, FOD was the more widely used strategy. However, since the 1990s, OOD has gained widespread popularity, and is regarded today, as the preferred approach for many scenarios. Whichever strategy is employed, the quality factors mentioned in [chapters 1](#) and [3](#) apply here. Quality must be built into the design from the outset. In the interest of clarity, these quality factors are restated here:

Efficiency, Reliability, Flexibility, Security, User-friendliness, Integrity, Growth Potential, Maintainability, Adequacy of Documentation, Functionality, Cohesiveness, Adaptability, Productivity, Comprehensive Coverage

9.2.1 Function-Oriented Design

In FOD, the system is designed from a functional viewpoint, starting at the highest level, and progressively refining this into a more detailed design. The software engineer explicitly specifies the “*what, wherefore and how*” of the system; subsequent development must also deal with the “*what, wherefore and how*” as separate issues to be tied together.

FOD commences with the development of DFDs or POFs and other function-oriented system flow charts mentioned in [chapter 6](#).

FOD conceals details of an algorithm in a function, but the system state (data) is not hidden. In fact, there is a centralized system state, shared by all functions. Further, changes to a given function can affect the behavior of other functions, and by extension, the entire system.

FOD leads to a system of interacting functions, acting on files and data stores (system state). Here, the principle of *data independence* (immunity of application programs to structural changes of an underlying database) is very important. As you will see later, violation of this principle could be catastrophic.

9.2.2 Object-Oriented Design

In OOD, the system is designed from an object-oriented viewpoint, and consists of *objects* that have hidden states (data) and methods; each object belongs to an *object type*. The software engineer explicitly specifies the “*what*” of the system (in the form of object types), but focuses on “*encapsulating the wherefore and how*” of the system into its objects.

This course recommends that your OOD should commence with the

development of an information topology chart (ITC) or object flow diagram (OFD), then employ other object oriented diagramming techniques (such as O-R diagrams, UML diagrams, transition diagrams, activity diagrams and other OO techniques) mentioned in [chapters 6](#) and [7](#).

OOD conceals details of an object by encapsulating these details in an object type, implemented as a class, but the behavior is not hidden. Moreover, changes to the internal structure of an object type (class) are isolated from all other system objects.

OOD leads to a system of interacting objects, each with their own internal structure and operations (interaction is facilitated through the operations). Here, the principle of *encapsulation* (information hiding) is very important. Encapsulation to the OO paradigm is what data independence is to the FO paradigm.

An *object type* is an entity that has a state (data structure) and a defined set of operations that operate on that state. The state is represented by a set of attributes, thus giving the object a structure. Object behavior is represented by operations, which are implemented by methods. The attributes and operations of an object (type) are referred to as *properties*. Object types are implemented as *classes* that encapsulate both data structure (attributes) and behavior (operations). These concepts are illustrated in [Figure 9-2](#); this figure shows the UML (Unified Modeling Language) representation of an object type called Employee. Such a diagram is called an *object structure diagram* (OSD), or simply a *class diagram*.

Employee // The name of the object type
// The name and data type of each data attribute of the object type
Employee# : String LastName: String FirstName: String DateOfBirth: Number ... Salary: Decimal
// The name of each operation with parenthesized parameters, and return type
Create(): Void Hire(): Void Modify(): Void Fire(): Void Interview(): Void Resign(): Void Promote(): Void Retire(): Void Remove(): Void Archive(): Void

Figure 9-2. UML Diagram for an Object Type

Objects communicate with each other via messages. Messages are usually implemented as procedure (or function or method) calls with appropriate parameters.

OOD facilitates *inheritance*:

- Every object must belong to a class from which it inherits all its *properties* (attributes and operations). In object-oriented environments, a class is also an object, and may therefore inherit properties from another class called the *super-class* (also called the *parent class* or *base class*). Inheritance hierarchies can therefore be established as illustrated in [Figure 9-3](#). The inheriting class is called the *sub-class*, *child class*, or *derived class*.

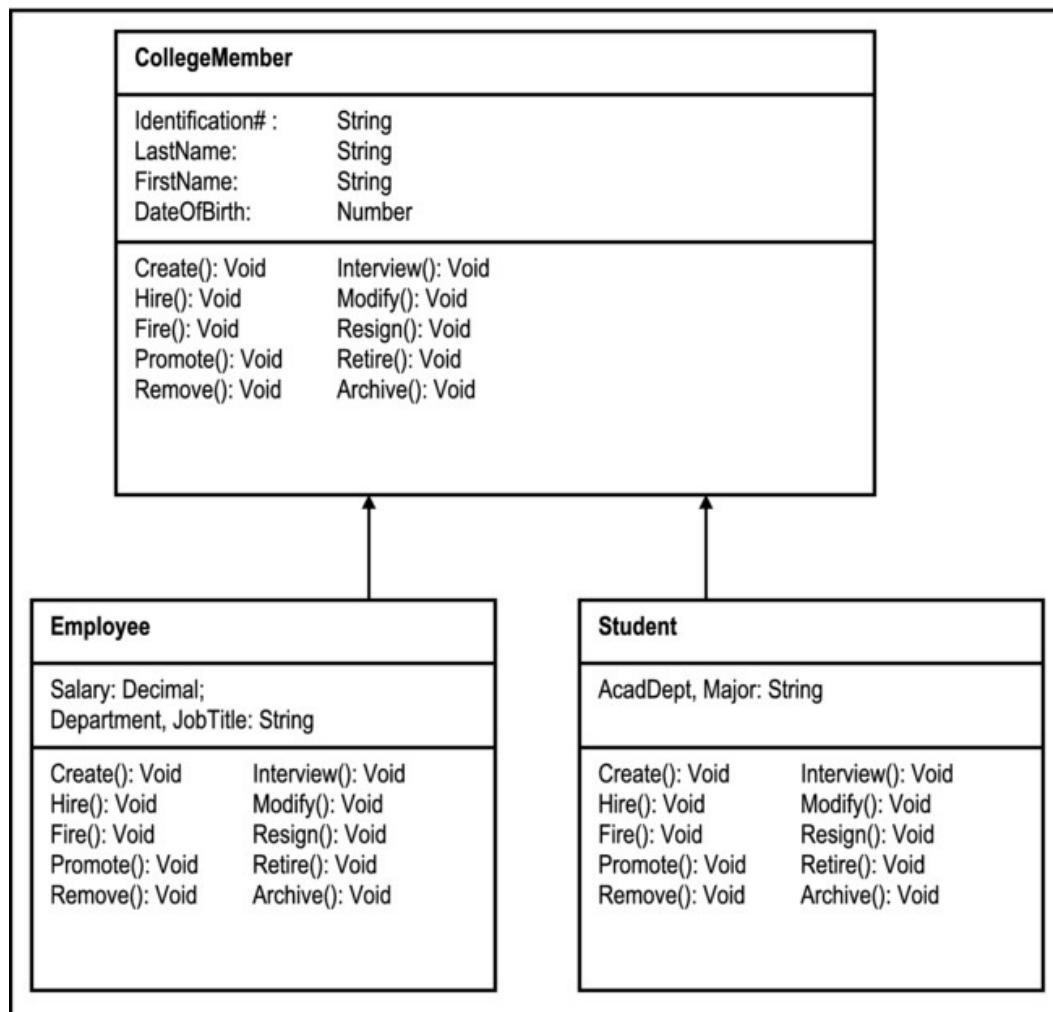


Figure 9-3. UML Diagram of a Type Hierarchy for a College Community

- A class could inherit properties from more than one super-class. This situation is referred to as *multiple inheritance* and is illustrated in [Figure 9-4](#). Multiple inheritance, while providing flexibility, can be a source of confusion; for this reason, it is avoided in certain environments (e.g. Java). In your course on OOM, you will learn about *workarounds*, to avoid this situation (see [appendix 5](#)).

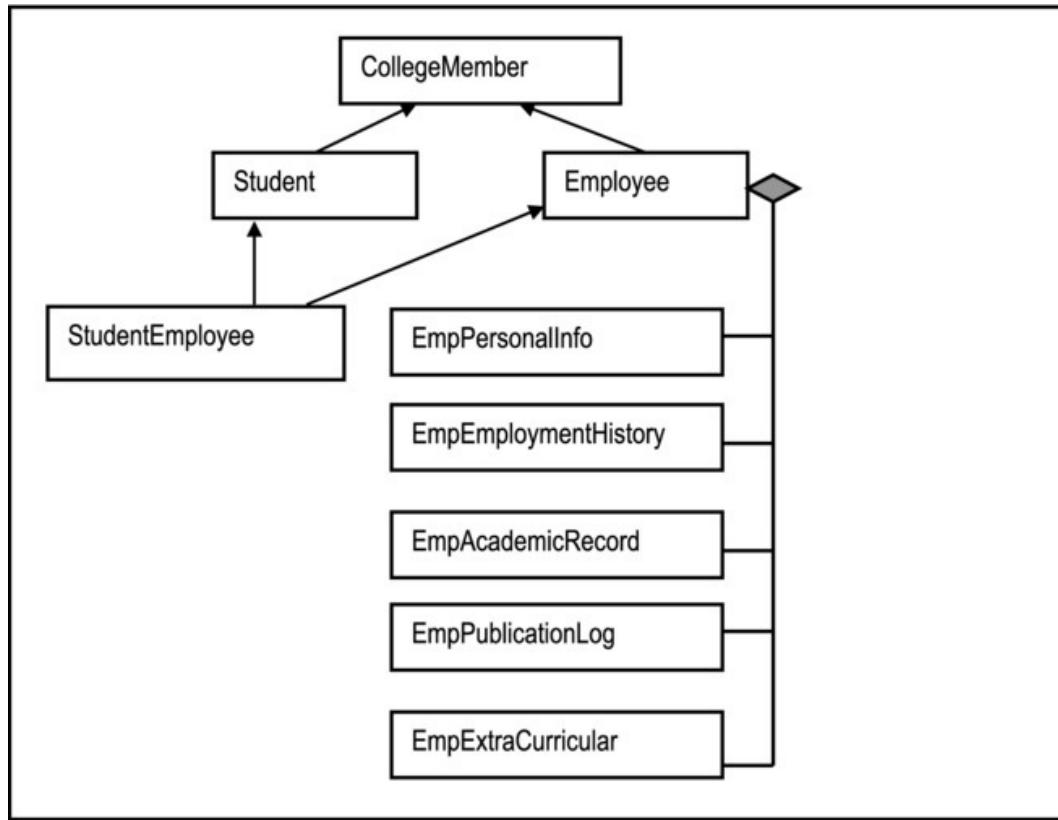


Figure 9-4. Abbreviated UML Diagram of an Inheritance Network for a College Community

- A class could be a super-class and a sub-class at the same time; this is also illustrated in [Figure 9-4](#).

OOD facilitates *amalgamation*: An object type may be defined as the amalgamation of several constituent object types. This is illustrated in [Figure 9-4](#), where **Employee** is the composition of **EmpPersonalInfo**, **EmpEmploymentHistory**, **EmpAcademicRecord**, **EmpPublicationLog**, and

EmpExtraCurricular. If the amalgamation is optional (i.e. the constituents can exist on their own), it is called an *aggregation*, and the diamond shape is not shaded in; if the amalgamation is mandatory (i.e. the constituents cannot exist on their own), it is called a *composition*, and the diamond shape is shaded in (as in the figure). Quite often, the distinction is not very clear, and the software engineer has to make a judgment call on the matter.

Another important concept in object technology is *polymorphism* — the act of an object or operation taking on a different form, depending on the prevailing circumstances. To illustrate, consider an operation, **Create()**, which creates a college member object (refer to [Figure 9-4](#)). This operation could be made to behave differently, depending on whether the object being created is a **CollegeMember**, an **Employee**, or a **Student**.

[Appendix 7](#) discusses OOM tools (please review); some of the more popular OO-RAD tools include Delphi, Rational Rose, and WebSphere; some of the more popular object oriented programming languages (OOPLs) include Java, C++, and Object Pascal (C++ and Object Pascal are really *hybrid languages* – supporting both procedural programming and OO programming). Note however, that OOD is independent of any programming language or OO-RAD tool used for software development. In fact, OOD is possible, even with traditional software development tools (though more effort would be required). It is also possible to do procedural programming in a hybrid programming environment such as C++ or Object Pascal.

By observation, most OO environments are actually hybrid environments, where an *OO user interface* is superimposed on a *relational database* (examples include DB2 and Oracle). This will be further elucidated later (sub-section 9.2.5).

9.2.3 The Unified Modeling Language

The Unified Modeling Language (UML) was developed by three leading prodigies in the area of object technology — James Rumbaugh, Grady Booch and Ivar Jacobson of Rational Software. This language was developed as an OO modeling language and is widely used in actual software development as well as research.

UML defines standards for object structure as well as object behavior ([Figure 9-3](#) and [Figure 9-4](#) employ some of the symbols used for object structure). You will be further exposed to this language in the upcoming chapters

(see the recommended readings also).

9.2.4 Advantages of Object Oriented Design

Object oriented design brings a number of advantages to the software engineering arena. Some of the commonly mentioned ones are as follows:

- Reusability of code
- Easier maintenance
- Enhancement of the understandability of the system
- Higher quality design
- Design independence — classes which are independent of platforms can be developed
- Large, complex systems are simplified to interacting objects
- More powerful OO CASE tools and RAD tools have evolved. For many of these tools, the SDLC is simplified, merging actual design and development into what we call *modeling*, since many of the diagrams are said to be *executable diagrams* (meaning that actual code is generated from the diagram).

9.2.5 Using Both FO and OO Strategies

In large and complex projects, FOD and OOD can be skillfully employed to be complimentary rather than competing. Borrowing principles from both strategies often leads to better software design.

OOD seems most natural and beneficial at the highest and lowest levels of system design. At the highest level, it is more convenient to perceive a system as a set of interrelating component subsystems (which can be implemented as super-classes). At the lowest level, objects and operations can be implemented as programming classes.

FOD seems most natural and beneficial at the intermediate level, where the system can be viewed as a set of interacting operations (implemented as programs). Also, remember that at some stage, methods for objects must be

specified; this often involves some amount of procedural programming.

Object technology does not make obsolete, all traditional (FO) approaches to software construction. Rather, it adds clarity, creativity and convenience to the software engineering discipline. In many hybrid environments, an OO user interface is often superimposed (made to access) a relational database. In fact, products such as Oracle, DB2, Delphi, Gupta Team Developer, etc. all reflect this approach.

One final note: FOD may or may not result in a change in the way users do their work. However, OOD usually results in a change in the way users do their work; and the users usually like the change.

9.3 Architectural Design

Complex systems are typically composed of subsystems or modules, which are themselves complete systems. These components must seamlessly integrate into the larger system. Architectural design addresses this challenge by determining what the components are and how they integrate and communicate with each other. The following are some considerations (factors) that influence system architecture decisions:

Performance: If performance is a critical requirement, the architecture should seek to minimize the number of components that have to cross-communicate (i.e. employ more *large-grain* components than *fine-grain* components).

Security: If security is critical, the architecture should provide a stringent security mechanism (preferably layered with the most critical resources at the innermost layer).

Availability: If availability is a critical requirement, the architecture should include controlled redundancies by making components as independent as possible.

Maintainability: If maintainability is a critical requirement, the architecture should employ fine-grain, self-contained components that can be readily changed.

These requirements typically do not synchronize with each other, so trade-offs will be necessary. Two issues of paramount importance in architectural design are *resource sharing* and *system control*. We will discuss these issues in the next two sub-sections, then close the section with a brief discussion on system components.

9.3.1. Approaches to Resource Sharing

There are four approaches to resource sharing:

- The Repository Model
- The Client-server Model
- The Abstract Machine Model
- The Component Model

Repository Model

In the repository model, all shared resources (data and operation) are stored in a central holding area (library). Each component has access to the repository. The repository may consist of more than one library of shared resources. It may contract or expand as the system is maintained during its useful life.

Examples of software systems that employ this approach include management information systems, CAD systems and CASE toolsets. [Figure 9-5](#) also provides, by way of example, an overview diagram of the CUAIS project (mentioned in earlier chapters), assuming a repository model. In this approach, the central CUAIS database and controller subsystem interfaces with all other subsystems.

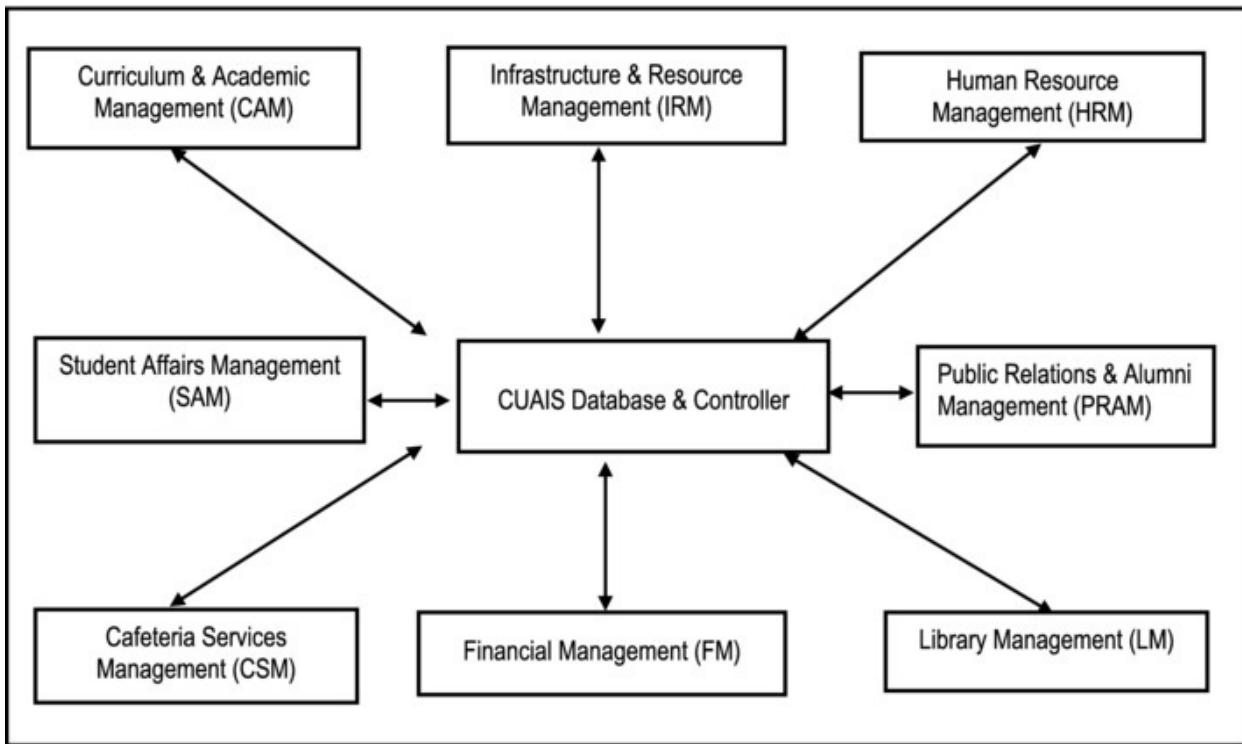


Figure 9-5. OFD for the CUAIS Project, Assuming Repository Model

Advantages of repository-based software include the following:

- Efficient data sharing; no need to explicitly transmit data for sharing
- High level of data independence: subsystems need not be concerned with how other subsystems use data
- Backup and recovery can easily be managed
- Integrating new resources is easy

Disadvantages of repository-based software include the following:

- Since subsystems have to agree on the data model, performance may be compromised.
- Evolution may be difficult, since the data model is standardized. Translating it to a new model can be costly.
- Flexibility in component subsystems may be lost on issues such as backup and recovery.

- It might be challenging to distribute the repository over a number of platforms.

Client-Server Model

In the client-server model, a set of independent network servers offers services which may be called on by members of a set of client systems. There is a server version of the software system that typically runs on a designated server machine, and a client version that runs on designated client machines. A network allows communication among clients and servers.

Examples of software systems that employ this approach include management systems with stand-alone components. Also, [Figure 6-7](#) (of [chapter 6](#)) provides a partial overview of the CUAIS project without the central database. It is repeated here (as [Figure 9-6](#)) for ease of reference. In this approach, each subsystem would operate independently, but with the capability of communicating with other subsystems comprising the system.

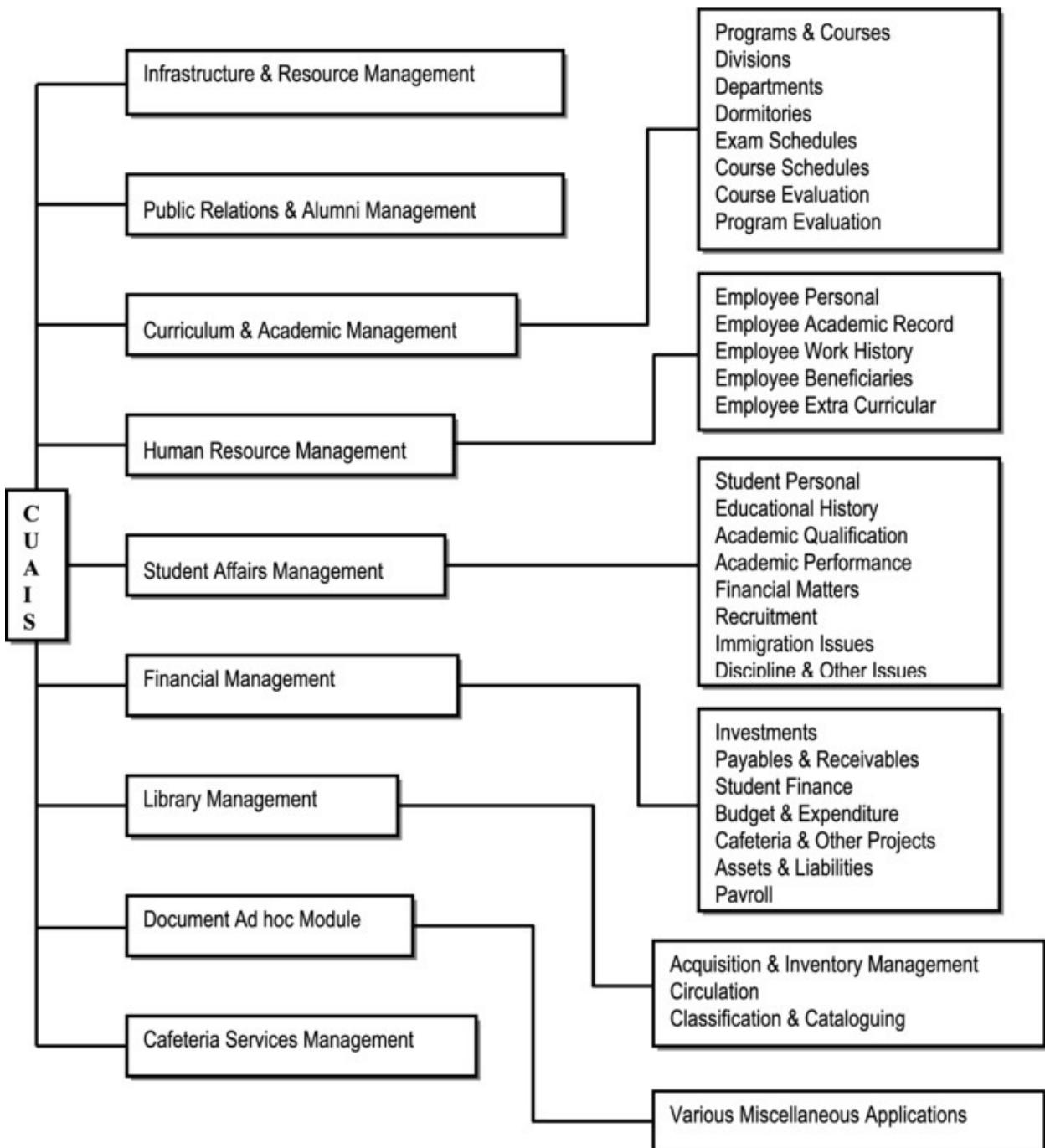


Figure 9-6. Partial Information Topology Chart for the CUAIS Project

Among the significant advantages of the client-server model are the following:

- A *distributed architecture* is facilitated. The implications of this are very profound and far-reaching, as you will discover

in other advanced courses (*electronic communication systems, database systems, and operating systems*).

- The system can easily grow with additional clients and/or servers.
- The approach provides flexibility in how components communicate (via request, data transfer or service transfer).
- Productivity is enhanced.

The model faces three major challenges:

- More sophisticated software are required (but this is readily available) for resource management and communication among the components.
- Each server must take responsibility for backup and recovery issues.
- In situations where data is replicated in the interest of efficiency, maintaining the integrity of the data is problematic.

Abstract Machine Model

In this model (also called the *layered model*), the software is organized into layers, each providing a set of services. Each layer communicates directly only to the layer immediately above or below it.

By way of example, the *open system interconnection model* (OSI model) for communications protocols typifies a layered approach. Additionally, some operating systems and compilers are designed using the layered model.

The main advantage of the layered approach is that problems can be easily isolated and addressed. The main drawback is that the approach is not relevant to all kinds of system problems.

Component Model

The component model describes a pre-client-server approach where a set of component systems resides on a given machine, typically in a network environment. Obviously, this scenario can be easily facilitated in a client-server

environment. But observe that it could also be implemented in a non-client-server environment, for example a minicomputer environment (precisely what used to happen prior to the introduction of client-server technology).

The component systems are made to communicate via interface programs and/or the introduction of controlled redundancy. To illustrate, suppose that **System-A** and **System-B** both use a database file, **FileX**. Let us assume that **System-A** owns **FileX** and therefore has update rights. **System-B** has a copy of **FileX**, namely **FileXp**, which it uses for data retrieval only. An interface program could be responsible for periodically copying **FileX** to **FileXp**, so that **System-B** “sees” accurate data.

Multi-component as well as single-user systems (on stand-alone machines) also typify the component model. The approach is relevant whenever a complex software system can be constructed by putting autonomous components together, or the system can be decoupled into identifiable autonomous components. [Figure 9-6](#) would therefore also be applicable if the CUAIS project was being constructed using the component approach.

Two significant advantages of component-based systems as described here are local autonomy and fast response. Two drawbacks are the potential data integrity problems that they pose, and their limited scope of applicability.

With advances in OOM, we have witnessed a resurgence of a revised component model, called component based software engineering (CBSE). As mentioned earlier in the course (review section 1.4), CBSE promises significant benefits to the software engineering discipline.

9.3.2 System Controls

System control relates to how executing components are managed. There are two approaches that you should be familiar with:

- Centralized control
- Event-based control

In the centralized control approach, one system component is designated as the system controller and has responsibility for execution of the other components. Two models are prevalent:

- The *call-return model* is relevant to sequential systems: A

component is invoked by a call; after execution, control goes back to the caller.

- The *manager model* is relevant to concurrent systems: One component controls starting, stopping and coordination of other components. Components may run sequentially or in parallel.

In the event-driven approach, the events control the execution of system components. Two models are prevalent:

- In a *broadcast model*, an event is broadcasted to all components. A component that can handle the event responds to it. *Ethernet* and *Token Ring* network protocols are good examples of the implementation of broadcast-driven control.
- In an *interrupt-driven model*, an interrupt handler responds to interrupts from various components and arbitrates which component will respond to the intercept. Operating systems software and real time systems are typically designed with interrupt handlers.

9.3.3 System Components

If the software engineering is in the FO paradigm, the system is broken down into a set of interacting functional components. If the OO paradigm is in vogue, the system is broken down into a set of interacting objects. In either case, consideration must be given the foregoing architectural design issues.

9.4 Interface Design

Very closely related to the architectural design is the system interface design (in fact, interface design may be incorporated into the architectural design). Here, the software engineer specifies exactly how the various system components will communicate.

In specifying how components will communicate, please note the following guidelines:

- Identify and specify all *utility operations* to be employed.
- Design intermediate data structures that may be required.
- Identify and specify system-wide utilities, for example date validation, error message processors, help retrieval processors, printing aids, etc.

9.5 Software Design and Development Standards

Software standards are very important in the design and development of software. The larger and more complex the project, the greater is the need for (and therefore benefit of) such standards.

Every software engineering company should have such standards; standards should also exist in companies where information technology (more specifically, the development of information systems) plays a critical role in the realization of corporate goals.

9.5.1 Advantages of Software Standards

The existence of software standards leads to a number of benefits, some of which are mentioned below:

- Enhancement of consistency in design and coding
- Enhancement of understandability of the system
- Enhancement of the maintainability of the system
- Improvement of the likelihood of reliable software
- Enhancement of good software documentation
- Imposition of some degree of order on software design and coding; programmers have to code to established standards, and not go off in unwieldy explorations

- Enhancement of a high level of efficiency and productivity during software construction
- Facilitation of a higher level of quality of the final product
- Facilitation of better project management

These advantages significantly contribute to the development of software systems of high quality. The converse is also true: failure to develop and observe meaningful software standards is a prescription for spaghetti code and mediocre software systems. It is through standards that the software engineering industry has been able to produce platform independent software systems that are immune to international borders and cultural barriers. Had there not been software standards, we would not have many things that we sometimes take for granted — the World Wide Web, operating systems, database systems, telephone services, television, radio, aviation, etc. In other words, without standards we'd all still be in the Stone Age, and software engineering as we know it would not exist.

9.5.2 Issues That Software Standards Should Address

The essential issues that software standards should address vary from one organization to another. They also vary with the category of software being developed. However, there are some fundamental ones that should always be treated. These are mentioned below:

Naming of Non-database (Hardware Related) Objects:

This includes the naming of workstations, printers, user profiles, output queues, etc. It includes all relevant objects related to the hardware used in the organization.

Naming of Database Objects: This activity includes the naming of physical database files (tables), logical views, database stored procedures and triggers, etc.

Naming of User Interface Objects: This includes the naming of operations (or application programs), menus and other user interface objects that may be employed in software construction. These objects will vary according

to the software development tool being used.

Database Design: Standards must be set in respect of database design. [Chapter 10](#) and your course in *database systems* will elucidate this area.

Screen and Report Layout: It is always a good habit to have standardized screen and report layouts for a given system. [Chapter 11](#) will provide further clarification.

Forms Design: Where the software requires manual inputs, or generates standard forms as output, these forms must be carefully designed. [Chapter 11](#) will elucidate this area.

User Interface Design: There should be guidelines with respect to the user interface (see [chapter 11](#)).

Operation Specification: This relates to the method used to specify the required operations of a system. [Chapter 12](#) will provide further clarification.

Programming standards: This includes generic coding approaches, use of special keyboard keys (e.g. function keys), error handling, etc. [Chapters 11 and 12](#) will provide further clarification.

Software Development Tools: It may be necessary to define guidelines and benchmarks for the use of certain software development tools.

System Documentation: There should be clear guidelines for software documentation in respect of system help, user manuals and other technical documents.

[Figure 9-7](#) provides an example of an object naming convention that has been used by the current author on several software engineering projects. The inventory management system (IMS) project of appendix10 applies or prescribes this convention for several of the above-mentioned areas. Additionally, a slight modification of this convention is used in [chapters 10](#) (Figure 10-9) and 12 (Figures 12-10 – 12-14).

Object Name: SSXXXXXX_MMn where
SS represents the system or subsystem abbreviation;
MMn represents the object mode or purpose (1-3 bytes);
XXXXXXXXXX represents the descriptive name of the object (6-8 bytes).

For example, valid subsystem abbreviations for the inventory management system (IMS) of earlier discussions may be:

AM: Acquisitions Management
FM: Financial Management
SC: System Controls

Valid mode abbreviations include:

BR: A base relation (if relational DB model)
OT: An object type (if OO DB model)
LVn: A logical view (e.g. LV1, LV2, etc.)
NXn: An index to a base table or object type (e.g. NX1, NX2, etc.)
PK: Primary Key
FKn: Foreign Key (e.g. FK1, FK2, etc.)
ICn: Integrity Constraint (e.g. IC1, IC2, etc.)
AO: An ADD operation
MO: A MODIFY operation
ZO: A DELETE (Zap) operation
IO: An INQUIRE operation
FO: A FORECAST operation.
RO: A REPORT operation
XO: A utility operation
DS: A database synonym or alias of a known database table
DC: A database constraint
DT: A database Trigger
DP: A database procedure or function
DK: A database package
MF: A Message file — a special purpose database table (file) to store the text (and other essential details) for diagnostic error and status messages

The descriptor used for a database base relation or object type is consistently used for other objects that directly relate to that object. For example, the objects used for the management of inventory items may be:

- AMInvMaster_BR — a base relation to store data on inventory items
- AMInvMaster_NX1 — an index on the base relation
- AMInvMaster_AO — an operation to ADD inventory items
- AMInvMaster_MO — an operation to MODIFY inventory items
- AMInvMaster_ZO — an operation to DELETE inventory items
- AMInvMaster_IO — an operation to INQUIRE on inventory items
- AMInvMaster_RO — an operation to REPORT on inventory items
- AMInvMaster_XO — a utility operation related to inventory items
- AMInvMaster_LV1 — a logical view of the base relation

Attribute implementation names are mere abbreviations of their more descriptive names.

Figure 9-7. Proposed Object Naming Convention

9.6 The Design Specification

~~Software Design~~

The *design specification* (DS) is another important deliverable for a software engineering project; it signals the end of the design phase (but remember, you may use a reversible life cycle model). Like the requirements specification, it is a formal, comprehensive document, providing further technical insights to the former. If the requirements specification is the initial blueprint of the software, the design specification is the final blueprint; it represents a refinement of the requirements specification, and contains information that is used for the construction of the software product.

 **Note** Some textbooks discuss one software blueprint in the form of a highly technical requirements specification. This course favors a less intimidating requirements specification as a precursor to a more detailed and technical design specification.

9.6.1 Contents of the Design Specification

The design specification includes details such as:

- Acknowledgements
- Introductory Notes
- System Overview
- Database Design Specification ([chapter 10](#))
- User Interface Design Specification ([chapter 11](#))
- Operations Design Specification ([chapter 12](#))
- Architectural Design Specification
- Product Documentation Specification ([chapter 13](#))
- Message Management Specification ([chapter 13](#))
- Software Development Standards (possibly as an appendage)
- Refined Schedule for Software Development and Implementation (review [chapter 8](#))

9.6.2 How to Proceed

You proceed to construct the design specification by using the requirements specification as your input. If the requirements specification is as accurate and comprehensive as it should be, then this is all you need. [Figure 9-8](#) provides basic guidelines, but please note that this is not cast in stone: your approach may vary in some areas, depending on the nature of the project. Also observe that this figure is a refinement of the lower portion of [Figure 7-10](#) (of chapter 7).

If at this point, you are still not confident about putting a design specification together, do not panic; after the next four chapters, and with practice, you will be in much better shape. The tools available for putting this deliverable together are very important. CASE tools (particularly OO-CASE tools) are excellent (review [chapters 1 and 2](#)). However, even in their absence, you can still be very effective with basic desktop processing tools.

Finally, please note that good software engineering will ensure that the requirements specification and the design specification both inform the final product documentation. Moreover, with experience, you will be able to work on both deliverables in parallel rather than in sequence (review section 9.1). An excellent product is a credit to excellent design, not a coincidence.

1. Refine the overview and introductory notes. Assuming the OOD approach, your overview should include a refined information topology chart and possibly an object flow diagram.
2. If there are no organizational standards for software design and development, develop standards for the project.
3. For each subsystem, provide the following:
 - Item 1
 - Database (or object structure) specification
 - Operations Specification
 - User Interface Specification
 - Message Specification
 - Documentation Specification
 - Security Specification
4. Refine the schedule for software development and implementation.
5. Conduct brainstorming sessions to verify the specifications.
6. Refine the introductory notes.
7. Prepare the acknowledgements.

[Figure 9-8. How to Construct the Design Specification](#)

9.7 Summary and Concluding Remarks

It is time once again to summarize what has been covered in this chapter:

- The software design process consists of architectural design, interface design, object structure design, operations design, user interface design, documentation design, message design, and security design.
- The design may proceed as FOD or OOD. OOD relies on OO methodologies, and is the preferred approach for contemporary software systems.
- Architectural design addresses the issue of integrating the various components that comprise the software system. It addresses issues such as resource sharing and system controls. Alternate strategies for resource sharing include the repository model, the client-server model, the abstract machine model, and the component model. Alternate strategies for system control include the centralized control and the event-driven control. In many cases, interface design is combined with architectural design.
- Software design and development standards are absolutely necessary if software products of a high quality are to be developed.
- The design specification (DS) is the software engineering deliverable that results from the software design process. This becomes the blueprint for the software system. The software engineer must be clear on what goes into the DS and how to prepare it.

If you desire to learn more about OOD, please refer to the appendices (as well as the recommended readings). In particular, [appendix 5](#) discusses categorizing object types in much more detail. For now, we must move on to other aspects of software design. The next few chapters discuss important components of the DS. As you proceed through these chapters, please reserve

the liberty to periodically examine [appendix 10](#) as this provides excerpts from a DS for the inventory management system of earlier mention.

9.8 Review Questions

1. Outline the software design process, explaining each aspect.
2. Compare function oriented with object-oriented design.
3. Discuss the four approaches to resource sharing as covered in the chapter. For each approach, cite advantages and disadvantages.
4. Examine [Figure 9-7](#). What conclusions can you draw about the system represented? Also examine [Figure 9-8](#). What conclusions can you draw about the system represented?
5. Discuss the importance of software design and development standards. Describe six important issues that these standards must address.
6. Which deliverable comes after software design? What is this deliverable comprised of? How would you proceed to construct such a deliverable?

9.9 References and/or Recommended Readings

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development With UML and Java*. Upper Saddle River, NJ: Prentice Hall, 2002. See [chapter 4](#).

[Martin, 1993] Martin, James, and James Odell. *Principles of Object-Oriented Analysis and Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1993. See [chapters 1 and 3](#).

[Peters, 2000] Peters, James F. and Witold Pedrycz. *Software Engineering: An Engineering Approach*. New York, NY: John Wiley & Sons, 2000. See [chapter 7](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapters 5 and 6](#).

[Pressman, 2005] Pressman, Roger. *Software Engineering: A Practitioner's Approach* 6th ed. Crawfordsville, IN: McGraw-Hill, 2005. See [chapters 9-11](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [chapters 7, 12 and 13](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See [chapters 11 – 14](#).

[Zhu, 2005] Zhu, Hong, *Software Design Methodology: From Principles to Architectural Styles*. Boston, MA: Elsevier, 2005.

CHAPTER 10



Database Design

If you review the OO modeling hierarchy (Figure 7-11) of [chapter 7](#), you will notice that the left hand side is characterized by the term *object structure analysis* (OSA). This chapter focuses on OSA, or more precisely, *object structure design*. In this chapter, we shall relax any distinction between *database design* and *object structure design*, for the following reason: As established in the previous chapter, irrespective of the software engineering paradigm employed, data (object) structure design is of paramount importance. By way of observation, most software engineering environments embrace the idea of a relational database, upon which an OO user interface is superimposed. This chapter presumes that convention, and provides you an overview of the database design experience. For a more comprehensive coverage of database systems, please refer to the recommended readings.

The chapter proceeds under the following captions:

- Introduction
- Approaches to Database Design
- Overview of File Organization
- Summary and Concluding Remarks

10.1 Introduction

A database is the record keeping component of a software system. Database design is critical part of software engineering. Underlying most software products is a database that stores data that is critical to the successful operation of the software. [Figure 10-1](#) provides you with some examples of this. In most

cases, the database is superimposed by a user interface, and may therefore sometimes not be obvious to the end user. However, whether it is obvious or not the database component is real and potent. A full discussion of database design is beyond the scope of this course; it is best done in a course in database systems. This chapter provides you with a useful overview of the territory.

Software Category	Database Need
Operating Systems	A sophisticated internal database is needed to keep track of various resources of the computer system including external memory locations, internal memory locations, free space management, system files, user files, etc. These resources are accessed and manipulated by active jobs. A job is created when a user logs on to the system, and is related to the user account. This job can in turn create other jobs, thus creating a job hierarchy. When you consider that in a multi-user environment, there may be several users and hundreds to thousands of jobs, as well as other resources, you should appreciate that underlying an operating system is a very complex database that drives the system.
Compilers	Like an operating system, a compiler has to manage and access a complex dynamic database consisting of syntactic components of a program as it is converted from source code to object code.
Information Systems	Information systems all rely on and manipulate internal databases, in order to provide mission critical information for organizations. All categories of information systems (DSS, EIS, MIS, WIS and SIS) are included.
Expert Systems	At the core of an expert system is a knowledge base containing cognitive data which is accessed and used by an inference engine, to draw conclusions based on input fed to the system.
CAD, CAM, and CIM Systems	A CAD, CAM or CIM system typically relies on a centralized database (repository) that stores data that is essential to the successful operation of the system.
Desktop Applications	All desktop applications (including hypermedia systems and graphics software) rely on resource databases that provide the facilities that are made available to the user. For example, when you choose to insert a bullet or some other enhancement in a MS Word document, you select this feature from a database containing these features.
CASE and RAD Tools	Like desktop applications, CASE and RAD tools rely on complex resource databases to service the user requests and provide the features used.
DBMS Suites	Like CASE & RAD tools, a DBMS also relies on a complex resource databases to service the user requests and provide the features used. Additionally, a DBMS maintains a very sophisticated meta database (called a data dictionary or system catalog) for each user database that is created and managed via the DBMS.

Figure 10-1. Illustrations of the Importance of Database

Database design is very crucial as it affects what data will be stored in and therefore accessible from the system. Hence, it affects the success of the system. Poor design leads to the following software flaws:

- Poor response time, hence
- Poor performance

- Data omissions
- Inappropriate data structures
- Redundancies
- Modification anomalies
- Integrity problems
- Lack of data independence
- Difficulty in system maintenance
- Inflexibility
- Lack of clarity
- Security and reliability problems
- Pressure on the programming effort to compensate for the poor design

Poor database design puts pressure on the software development team to program its way out of the poor design. By contrast, good design leads to the exact opposite of these conditions induced by poor design.

Some objectives of database design include the following:

- Comprehensive data capture
- Efficiency
- Flexibility
- Reliability
- Control of Redundancy
- Security and Protection
- Consistency and Accuracy
- Ease of access and ease of change
- Availability of information on demand
- Desirable data integrity
- Data independence — immunity of application programs to structural, storage or hardware changes of the database

- Clarity and multi-user access

10.2 Approaches to Database Design

There are two approaches database design:

- Conventional files
- Database approach which includes
 - Relational model
 - Object-oriented model
 - Hierarchical model
 - Network model
 - Inverted List model

10.2.1 Conventional Files

[Figure 10-2](#) illustrates the idea of conventional file approach. Application programs exist to update files or retrieve information from files.

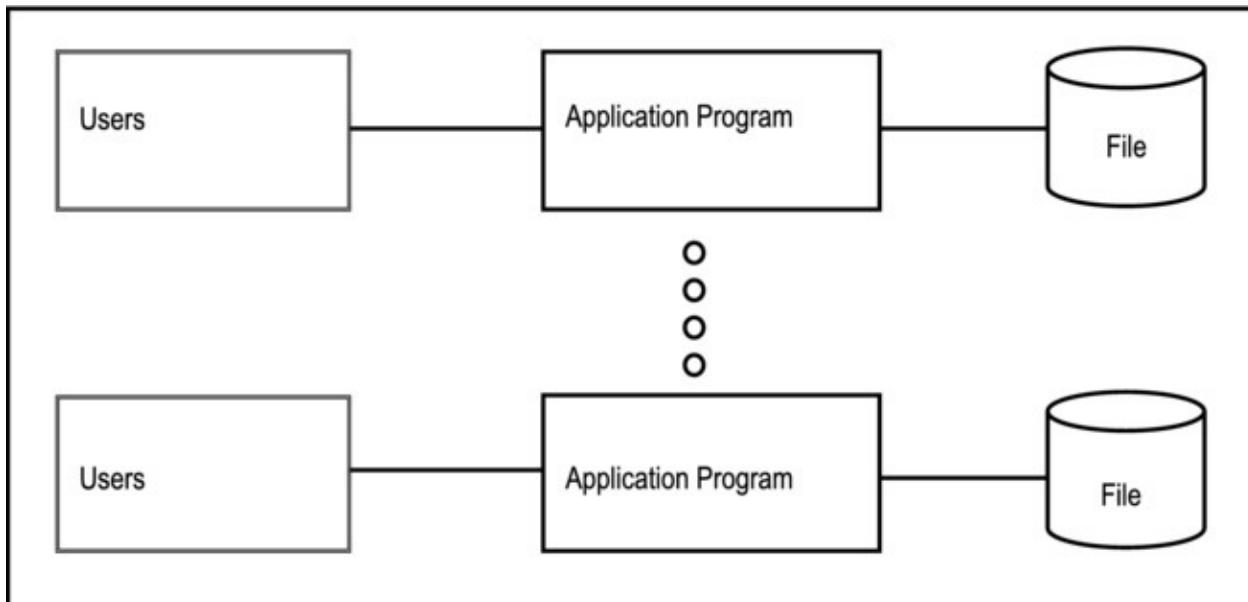


Figure 10-2. Conventional File-Based Design

This is a traditional approach to database design that might still abound in very old legacy systems (to be discussed in [chapter 18](#)). You may use this approach if the software system is already designed using this approach, and the task is to maintain it.

Note Do not attempt to redesign the software system without management consent. Also, be aware that people (including managers) sometimes get annoyed with a software engineer who walks around looking for every problem to fix. Ironically, fixing problems often created by human ineptitude or limitations is an integral part of your job. Just be discreet in the execution of your job ([chapters 14](#) and [15](#) provide more guidelines on how to conduct yourself on the job).

10.2.2 Database Approach

In the database approach, a database is created and managed via a database management system (DBMS) or CASE tool. A user interface, developed with an appropriate application development software, is superimposed on the database, so that end users access the system through the user interface. [Figure 10-3](#) illustrates the basic idea.

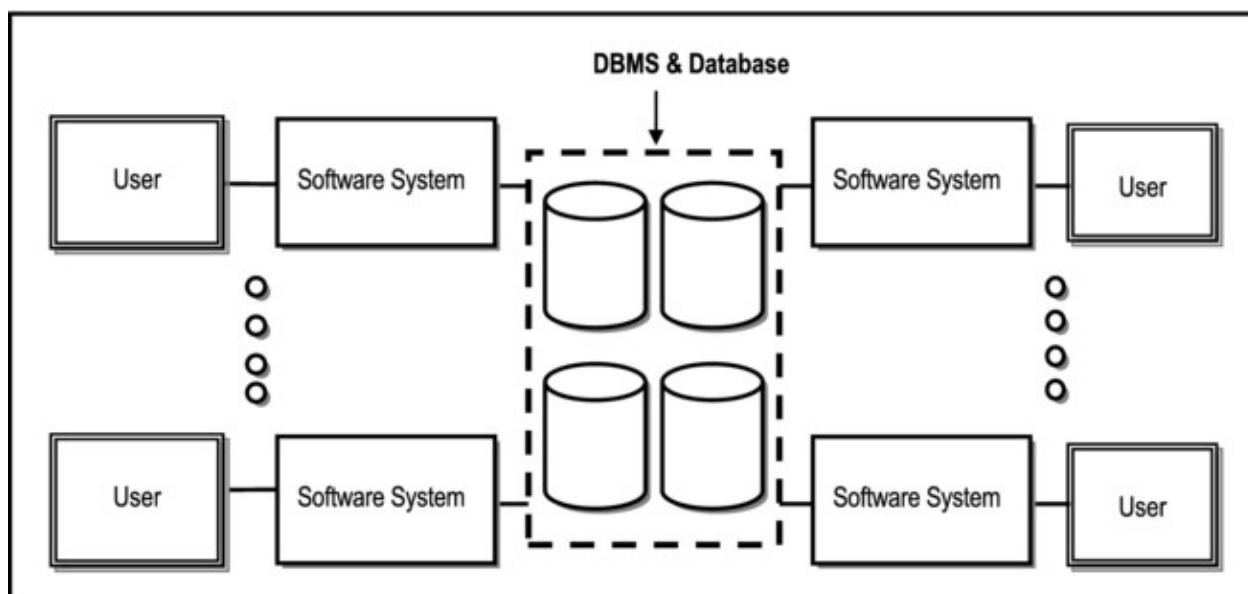


Figure 10-3. Database Approach to Design

Of the five methodologies for database design, the relational model and the OO model are the two which dominate contemporary software engineering; this is expected to continue into the foreseeable future. The other three approaches are traditional (from the 1960s and 1970s), but occasionally show up in legacy systems. They will not be discussed any further; for more information on them, see the recommended readings.

Following are the steps involved in designing a relational or OO database:

1. Identify data entities or object types
2. Identify relationships
3. Eliminate unnecessary relationships
4. Develop an *entity-relationship diagram* (ERD) or an *object-relationship diagram* (ORD)
5. Prepare the database specification
6. Develop and implement the database

Note Step 6 belongs to the field of database systems and will not be explored any further in this course. However, you should appreciate the close nexus between database design and software engineering.

10.2.3 Identifying and Defining Entities or Object Types

Identifying information entities or object types requires skills, experience, and practice. In this regard, the techniques discussed in [Chapter 5](#) are applicable. As you will recall, an information entity is a concept or object about which data is (to be) stored. An object type is concept or object about which data is (to be) stored, and upon which a set of operations are defined. If you relax the latter part of the definition of an object type, you will readily see that information entities and object types are similar.

Of equal importance is the structure of each information entity (object type). We define structure in terms of attributes: An entity (object type) is made up of

attributes (also called elements or properties) that describe the entity (object). Attributes are non-decomposable (atomic) properties about the entity (object), as illustrated in the example below ([Figure 10-4](#)).

Entity	Attributes
Department [MSPDEPT]	Dept Number [DeptNo, N4], Department name [DeprName A30]
Location [MSPLOCN]	Location [LocCode, A7], Location Name [LocnName, A30] , Distance from Head Office [LocDist, N5,2]
Employee [MSPEMPL]	Employee Number [EmpNbr, N7], Employee name [EmpName, A15], Date of Birth [EmpDOB, N7], ...
Projects [MSPPROJ]	Project Number [ProNbr, N5], Project name [ProNam, A30] ...
Part [MSPPART]	Part Number [PartNbr , A8], Part name [PartName, A20], Part unit [PartUnit, N5,2] ...
Supplier [MSPSUPLR]	Supplier Number [SuplNbr, A7], Supplier name [SuplName, A30]
Warehouse [MSPWHOUS]	Warehouse Number [WHNbr, A3], Warehouse name [WHName, A30], Warehouse size [WHSize, N5,2] ...

Note:

- It is a good habit to indicate the proposed system name (based on an established naming convention) for each database object (in the figure, system names for data entities are indicated in square brackets).
- For each attribute, assign a system name as well as some indication of the physical characteristics of the attribute. In the figure, this information is also enclosed in square brackets next to the respective attributes. The simple convention used here, is N for numeric data and A for alphanumeric data, with the length and decimal positions indicated.
- For a more comprehensive coverage, it is advisable to allow for a row for each attribute, so that additional information about the attribute can be specified.
- The process can be automated.

[Figure 10-4](#). Partial Entity-Attributes List for a Manufacturing Environment

10.2.4 Identifying Relationships

Identifying relationships among entities (object types) also requires skills, experience and practice, but can often be intuitively recognized by observation. To identify relationships, you have to know what a relationship is and what types of relationships there are. Your course in database systems will elucidate these issues to some level of detail. For now, you may consider a relationship as a mapping involving two or more information entities (or object types) so that a data item (an object) in one relates in some way to at least one data item (object) in the other(s) and vice versa. There are seven types of relationships:

- One-to-one (1:1) relationship
 - One-to-many (1:M) relationship
 - Many-to-one (M:1) relationship
 - Many-to-many (M:M) relationship
- }
- Traditional relationships
- Component relationship (if OO database)
 - Aggregation relationship (if OO database)
 - Super-type-sub-type relationship (if OO database)

The first four types of relationships are referred to as traditional relationships because up until object model (for database design) gained preeminence, they were essentially the kinds of relationships that were facilitated by the relational model. Observe also, that the only difference between a 1:M relationship and a M:1 relation is a matter of perspective; thus, a 1:M relationship may also be described as a M:1 relationship (so that in practice, there are really three types of traditional relationships). Put another way:

If E1, E2 are two information entities (or object types) and there is a 1:M relationship between E1 and E2, an alternate way of describing this situation is to say that there is a M:1 relationship between E2 and E1.

For traditional relationships, to determine the type of relationship between two entities (object types) E1 and E2, ask and determine the answer to the following questions:

- How many data items (objects) of E1 can reference a single data item (object) of E2?
- How many data items (objects) of E2 can reference a single data item (object) of E1?

To test for a component relationship between any two relations (object types) E1 and E2, ask and determine the answer to the following questions:

- Is (a data item of) E1 composed of (a data item of) E2?
- Is (a data item of) E2 composed of (a data item of) E1?

For a subtype relationship the test is a bit more detailed· for entities (or

For a subtype relationship, the test is a bit more detailed, for entities (or object types) E1 and E2, ask and determine the answer to the following questions:

- Is (a data item of) E1 also a (a data item of) E2?
- Is (a data item of) E2 also a (a data item of) E1?

The test is identical for object types, except that in the object-oriented paradigm, the term “instance” is preferred to “data item.” Possible answers to the questions are always, sometimes, or never. The possibilities are shown in [Figure 10-5](#).

Possibility	Implication
E1 always E2, E2 always E1	E1 and E2 are synonymous
E1 always E2, E2 sometimes E1	E1 is a subtype of E2
E1 always E2, E2 never E1	Makes no sense
E1 sometimes E2, E2 always E1	E2 is a sub-type of E1
E1 sometimes E2, E2 sometimes E1	Inconclusive
E1 sometimes E2, E2 never E1	Makes no sense
E1 never E2, E2 always E1	Makes no sense
E1 never E2, E2 sometimes E1	Makes no sense
E1 never E1, E2 never E1	No subtype relationship exists

[Figure 10-5.](#) Testing for Sub-type Relationship

Example 1: Again using the sample manufacturing subsystem of [Figure 10-4](#), several relationships can be identified, as indicated in [Figure 10-6](#).

Name of Relationship	Participating Entities	Type	Optional or Mandatory
Supplies	Suppliers, Parts	M:M	M
P-uses	Projects, Parts	M:M	M
Assigned	Projects, Employees	1:M	M
Belongs	Employee, Department	M:1	M
Hosts	Location, Department	1:M	M
Situated – W	Warehouse, Location	1:1	M
Situated – S	Supplier, Location	1:M	M
SPJ	Supplier, Part, Project	M:M	O
Composed of	Part, Part	M:M	M
Stocks	Warehouse, Part	M:M	M

[Figure 10-6.](#) Relationships List for a Manufacturing Environment

10.2.5 Developing the ERD or ORD

An entity-relationship diagram (ERD or E-R diagram) is a graphical illustration of entities and their relationships in the database. In the OO paradigm, the equivalent diagram is called an object-relationship diagram (ORD or O-R diagram). For small and medium sized projects, it is a very useful modeling technique. However, as the size and complexity of the system increases, the ORD/ERD tends to become unwieldy. In these circumstances, unless the software engineering team is using a CASE tool that facilitates generation and maintenance of the ERD/ORD, other pragmatic approaches are recommended. One such approach is to tabulate as illustrated above. Another approach is to construct for each information entity (object type), an object/entity specification grid (O/ESG). This will be discussed shortly.

The symbols used in an ERD are as shown in [Figure 10-7](#). [Figure 10-8](#) shows the ERD for the manufacturing system of [Figure 10-4](#), using the Crows-foot notation. In the diagram, the convention to show attributes of each entity has been relaxed. Note also that relationships are labeled as verbs so that in mapping one entity (or object type) to another, one can read an object-verb-object formation. If the verb is on the right or above the relationship line, the convention is to read from top-to-bottom or left-to-right. If the verb is on the left or below the relationship line, the convention is to read from bottom-to-top or right-to-left.

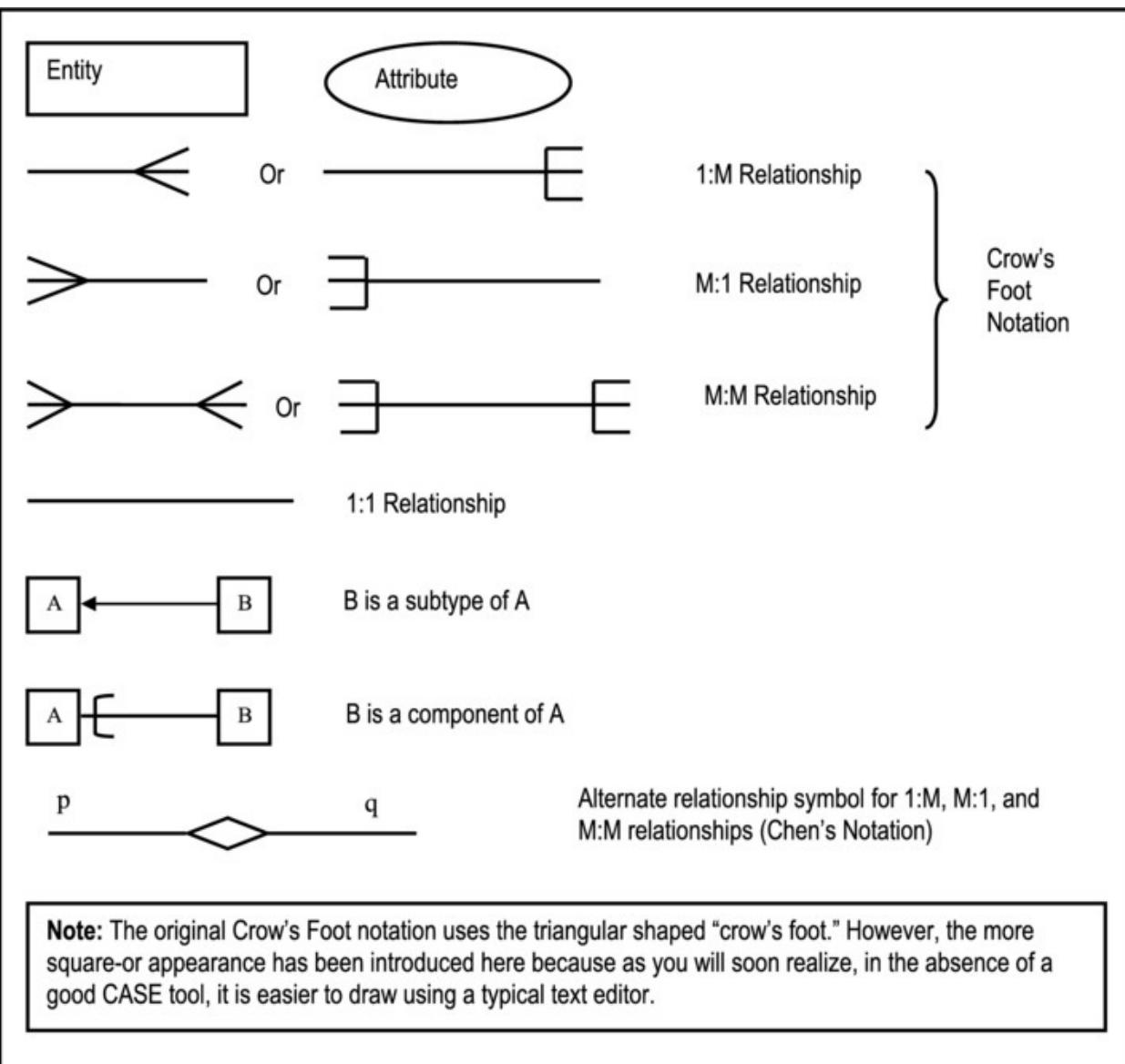


Figure 10-7. Symbols Used in E-R Diagrams

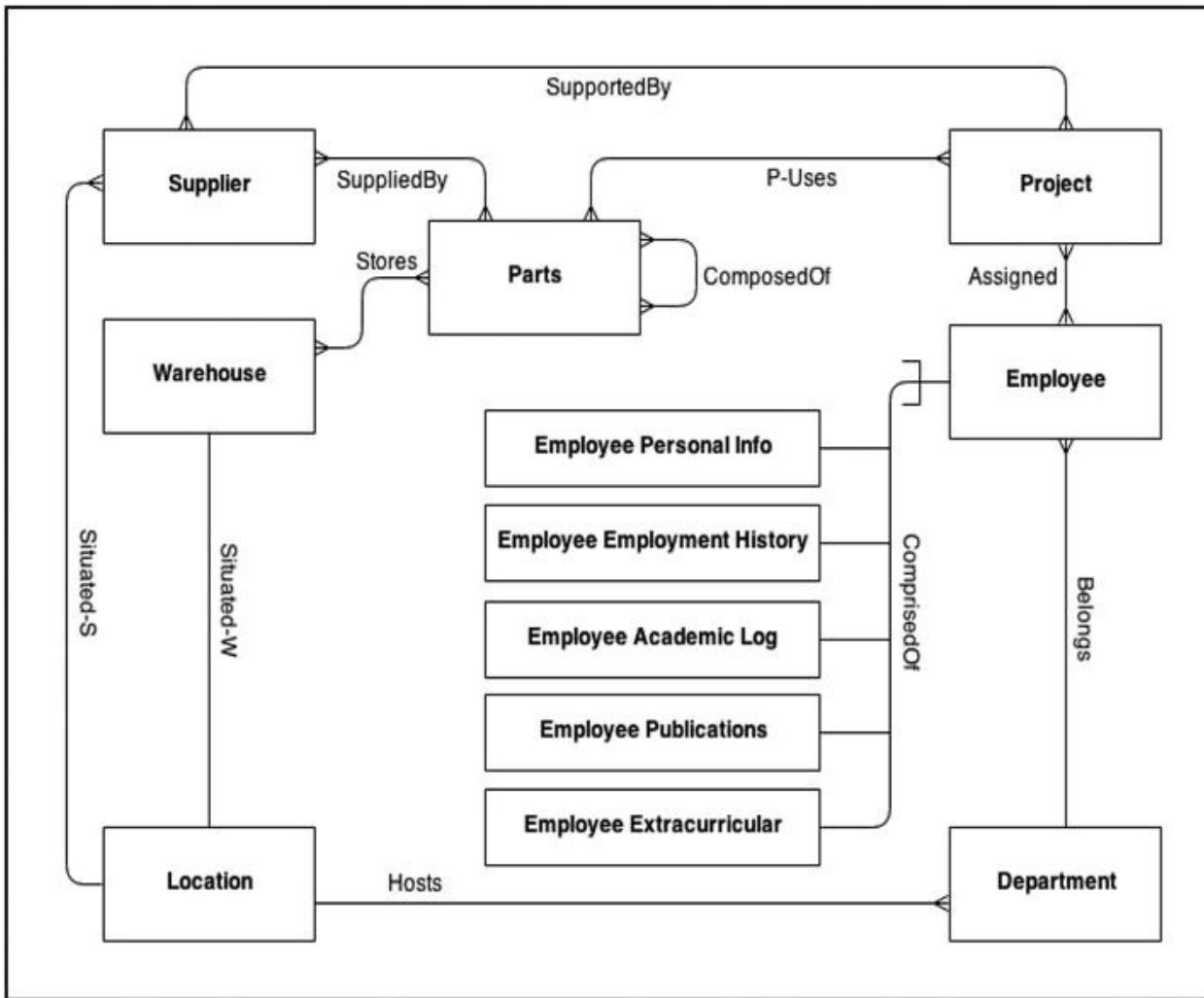


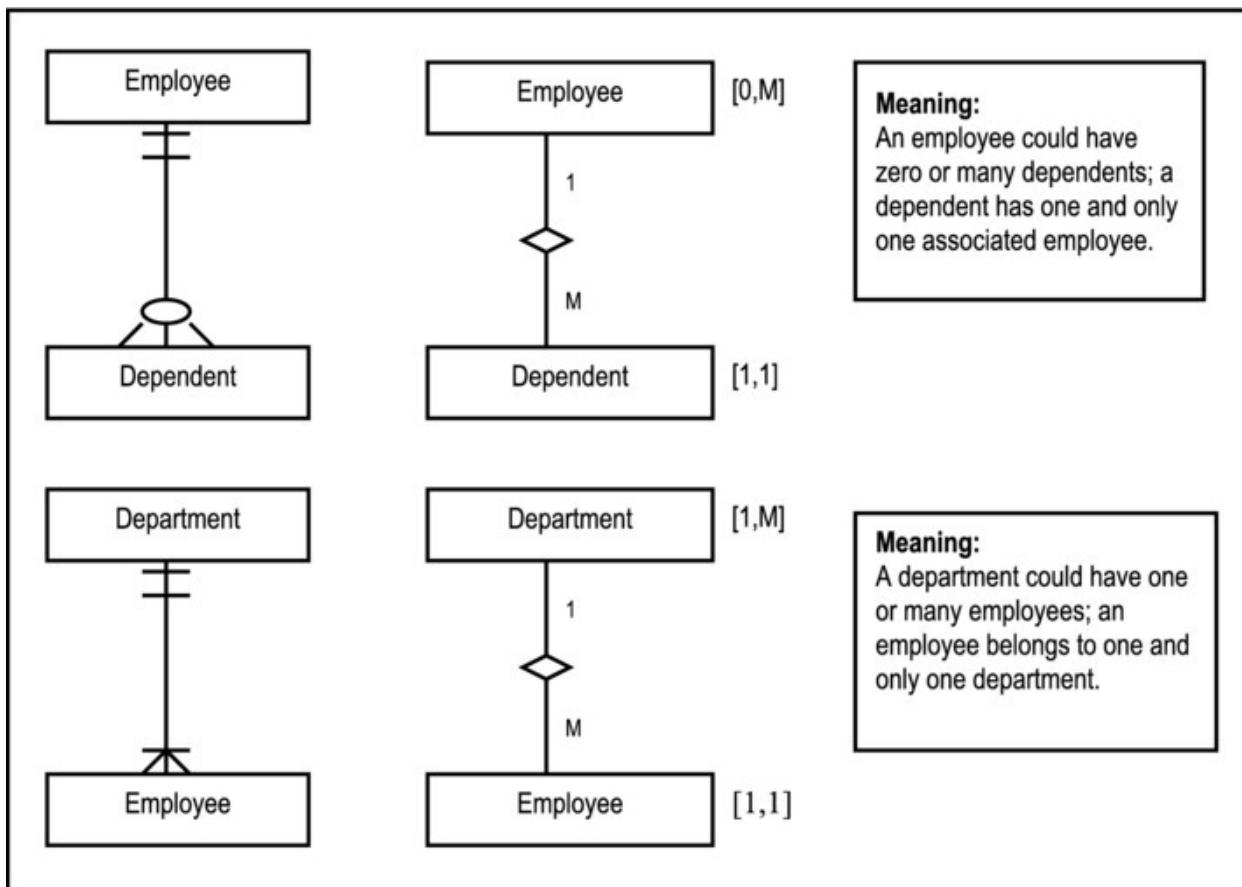
Figure 10-8. E-R Diagram for Manufacturing Environment

It is customary to indicate on the ERD, the multiplicity (also called the cardinality) of each relationship. By this we mean, how many occurrences of one entity can be associated with one occurrence of the other entity. This information is particularly useful when the system is being constructed. Moreover, violation of multiplicity constraints could put the integrity of the system in question, which of course is undesirable. Usually, the DBMS does not facilitate enforcement of multiplicity constraints at the database level. Rather, they are typically enforced at the application level by the software engineer.

Several notations for multiplicity have been proposed, but the Chen notation (first published in 1976, and reiterated in [Chen, 1994]) is particularly clear; it is paraphrased here: Place beside each entity, two numbers [x,y]. The first number (x) indicates the minimum participation, while the second (y) indicates the maximum participation.

An alternate notation is to use two additional symbols along with the Crow's Foot notation: an open circle to indicate a participation of zero, and a stroke (|) to indicate a participation of 1. The maximum participation is always indicated nearest to the entity box.

The Chen notation is preferred because of its clarity and the amount of information it conveys. [Figure 10-9](#) provides an illustrative comparison of the two notations.



[Figure 10-9](#). Illustrating Multiplicity Notations

Let us now turn our attention to the O-R diagram. The symbol for object type (mentioned in [chapter 9](#)) is also used in the ORD, and replaces the entity symbol (as you will soon see, they are actually similar). Assuming the UML notation, the following guidelines apply:

- Similar to the information entity, a box (square or rectangle) represents an object type. The object-type box has two additional compartments: one for attributes, and the other for

operations.

- A triangle or arrowhead (pointing towards the super-type) represents an inheritance relationship.
- An open diamond represents an aggregation relationship (the parts existing independent of the whole).
- A filled in diamond represents a composition relationship (the parts only exist as part of the whole).
- A line connecting two object types represents a traditional relationship; the *multiplicity* (also called cardinality) of this relationship is indicated by a pair of integers next to each object type; the lower value is indicated first, and an asterisk is sometimes used to mean “many.” The multiplicity of a relationship is the level of participation of each object type (or entity) in the relationship. The role that each object type plays in the relationship is also indicated next to the object type.

Since the object symbol automatically incorporates object attributes, there is therefore no need for an attribute symbol. In any event, including attributes and operations on the ORD tends to clutter the diagram rather quickly. It is therefore recommended that you omit this detail from the diagram for very large and/or complex systems. The upcoming subsection will suggest a creative and elegant way to represent attributes and related operations for object types comprising a system.

[Figure 10-10](#) illustrates an ORD (using UML notation) depicting aspects of a college environment. According to the diagram, **Student** and **Employee** are subtypes of **College Member**. Additionally, **Employee** is a composition of **Employee Personal Info**, **Employee Work History**, **Employee Academic Log**, **Employee Publication**, **Employee Extra Curricular**, and **Employee Dependents Log**.

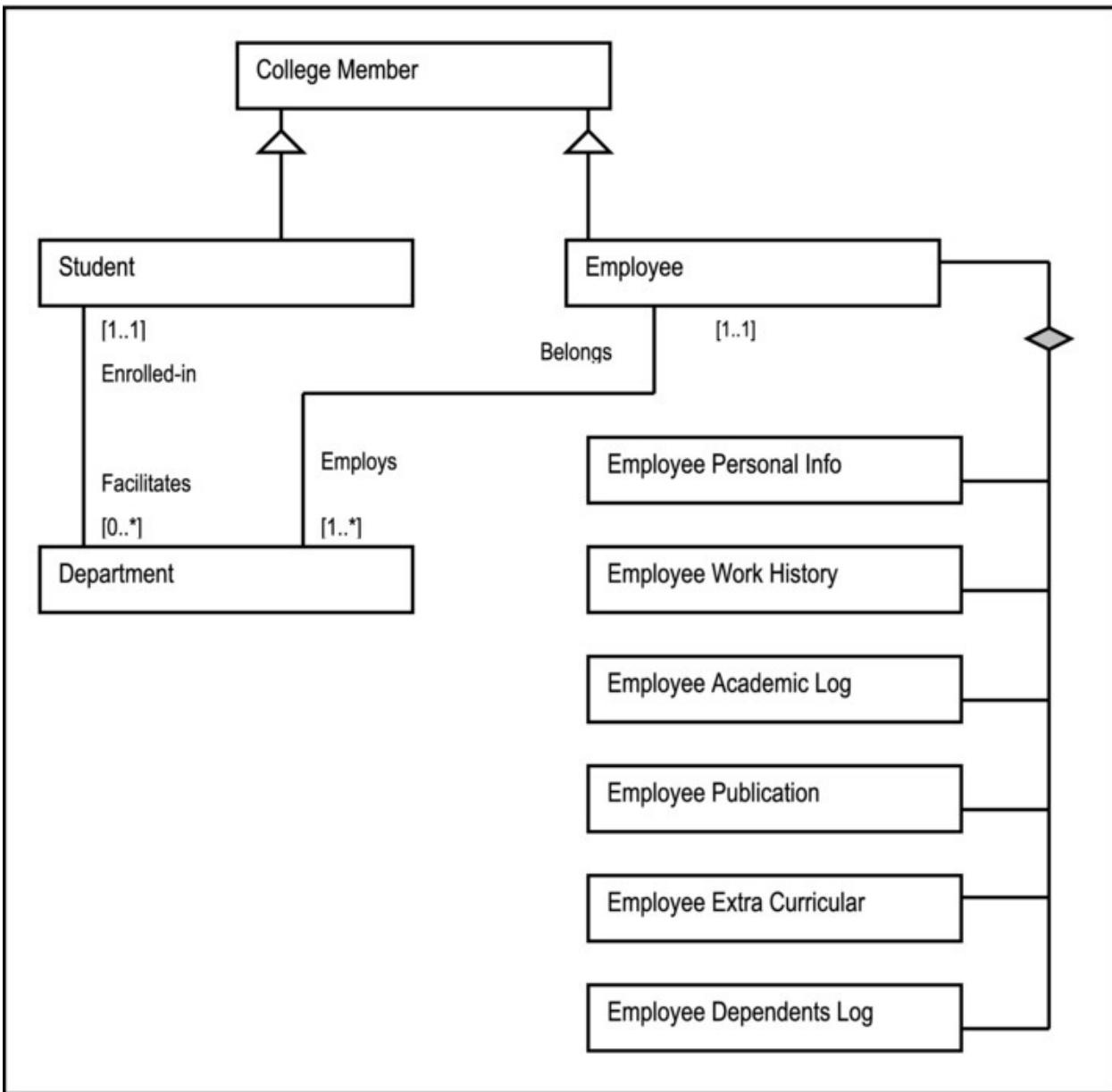


Figure 10-10. Partial O-R Diagram for College Environment (UML Notation)

Once, created, the ERD/ORD (or its equivalent) must be maintained for the entire life of the software system. If you are fortunate to be using a sophisticated CASE tool that automatically generates the diagram from the current database, then this will not be a problem for you.

10.2.6 Preparing the Database Specification

The database specification may be in different forms, depending on the available

resources. In an OO environment where you have the use a CASE tool that supports UML, it may simply be a detailed ORD where each object type is represented as explained and illustrated in section 9.2.2 of the previous chapter. In an FO environment, it may simply be a detailed ERD where the attributes of each entity are included on the diagram.

For large, complex projects (involving huge databases with tens of information entities or object types), unless a CASE or RAD tool which automatically generates the ERD/ORD is readily available, manually drawing and maintaining this important aspect of the project becomes virtually futile. In such cases, an *object/entity specification grid* (O/ESG) is particularly useful. The grid contains the following components:

- Descriptive name of the entity (object type)
- Implementation name of the entity (object type) — typically indicated in square brackets
- Reference identification for each entity, to facilitate easy referencing
- Descriptive name, implementation name (in square brackets) and characteristics (in square brackets) for each attribute
- References (implying relationships) to other entities in the system (indicated in curly braces)
- Comments on the entity and selected attributes
- Indexes (including primary key or candidate keys) to be defined on the entity
- Operations to be defined on each entity (object type)
- Optionally, implementation names of operations are be indicated in square brackets next to respective operations

The convention for specifying attribute characteristics is to use a letter to represent the nature of the data (A for alphanumeric, N for numeric and M for memo) followed by numbers representing the length and precision (for decimals). Figure 10-11 provides an illustration of an O/ESG for the manufacturing environment, or the college/university environment of earlier discussions. The ESG for three entities are included in the figure. In actuality, there would be one for each entity (object type) comprising the system. Also

note the special data attributes that reference other entities in the figure (E1.3, E2.4, E2.8). In order to determine when to introduce such references, you need to apply principles of database design. These principles are best discussed in a course on database design (see the recommended readings).

E1 – Department [HRDepartment_BR]**Attributes:**

1. Department Number [DeptNo] [N4]
2. Department Name [DeptName] [A35]
3. Department Head Employee Number [DeptHead] [N7] {Refers to E2}

Comments:

This table stores definitions of all departments in the organization.

Indexes:

1. Primary Key: [1] – constraint HRDepartment_PK
2. HRDepartment_NX1 on [2]

Valid Operations:

1. Maintain Departments [HRDepartment_MO]
 - 1.1 Add Departments [HRDepartment_AO]
 - 1.2 Update Departments [HRDepartment_UO]
 - 1.3 Delete Department [HRDepartment_ZO]
2. Inquire on Departments [HRDepartment_IO]

E2 – Employee [HREmployee_BR]**Attributes:**

1. Employee Identification Number [EmpNo] [N7]
2. Employee Last Name [EmpLName] [A20]
3. Employee First Name [EmpFName] [A20]
4. Employee Middle Initials [EmpMInitl] [A4]
5. Employee Date of Birth [EmpDOB] [N8]
6. Employee's Department [EmpDepNo] [N4] {Refers to E1}
7. Employee Gender [EmpGender] [A1]
8. Employee Marital Status [EmpMStatus] [A1]
9. Employee Social Security Number [EmpSSN] [N10]
10. Employee Classification Code [EmpClass] [A3] {Refers to E3}

....

Comments:

This table stores standard information about all employees in the organization.

Indexes:

1. Primary Key: [1] – constraint HREmployee_PK
2. HREmployee_NX1 on [2, 3, 4]
1. HREmployee_NX2 on [7]

Valid Operations:

1. Manage Employees [KREmployee_MO]
 - 1.1. Add Employees [HREmployee_AO]
 - 1.2. Update Employees [HREmployee_UO]
 - 1.3. Delete Employees [HREmployee_ZO]
2. Inquire on Employees [HREmployee_IO]

E3 – Employee Classification [HRClassif_BR]
Attributes:
1. Classification Code [ClsCode] [A3] 2. Classification Description [ClsDesc] [A30]
Comments:
This table stores definitions of all employee classifications.
Indexes:
1. Primary Key: [1] – constraint HRClassif_PK 2. HRClassif_NX1 on [2]
Valid Operations:
1. Maintain Classifications [HRClassif_MO] 1.1 Add Classifications [HRClassif_AO] 1.2 Delete Classifications [HRClassif_ZO] 2. Inquire on Departments [HRDepartment_IO]

Figure 10-11. Sample Object/Entity Specification Grid for Manufacturing Environment

10.3 Overview of File Organization

In planning the underlying database for a software system, it is important that the software engineer understands the different types of file organization techniques and the rationale and benefits of each. From your earlier programming courses, you should recall that there are four types of file organization:

- Sequential File organization
- Relative (direct) File Organization
- Indexed Sequential File Organization
- Multi-Access File Organization

Let us pause to briefly review each technique.

10.3.1 Sequential File Organization

In a sequential file, records are arranged in arrival sequence usually stored on systematic tape. Accessing of records must also be done sequentially (the file

may be sorted in a particular order).

Sequential file organization is useful when a large volume of records is to be added or updated in bulk or batch mode. [Figure 10-12](#) illustrates what a sequential file of student records may look like. Accessing the Nth record means first accessing N-1 records.

	Student ID	Last Name	First Name	Address
1	93010101	Foster	Bruce	Fox Lane ...
2	93060101	Ming	Rose	Rose Lane ...
3	92120101	Mano	Howard	Mano Lane ...
4	91010101	Henry	Adrian	Abbey Court ...
...				
n	99120101	Foxley	Sharon	Fox Lane ...

[Figure 10-12](#). Illustration of Sequential File

Sequential file organization is not suited for interactive processing. This is so because records have to be accessed sequentially in arrival sequence. If you have a file with thousands of records, attempt to provide interactive processing on a sequential file would produce very poor results, and would therefore be counterproductive.

10.3.2 Relative or Direct File Organization

In relative (direct) file organization, records are arranged in some logical order where there is a relationship between the key used to identify a particular record and the record's address on the storage medium. In computer science, this is often represented as follows:

F(key) ► Address

Use of linked list is one method of implementing relative file. Each record has a pointer to the next logical record. Access is improved but additional data has to be stored with each record. [Figure 10-13](#) illustrates a relative file implemented by linked list (on surname). The main problem with this implementation is that only sequential access is facilitated.

	Student ID	Last Name	First Name	Address	Last-Name Pointer
1	93010101	Foster	Bruce	Fox Lane ...	4
2	93060101	Ming	Rose	Rose Lane ...	END
3	92120101	Mano	Howard	Mano Lane ...	2
4	91010101	Henry	Adrian	Abbey Court ...	3
...					

Figure 10-13. Illustration of Linked List (Relative File)

Other more desirable methods of implementation of relative files are direct mapping, table lookup, hash functions, and open addressing with buckets (you should be familiar with these methods from your course in data structures and algorithms). These methods facilitate random (direct) access of the file(s). Consequently, interactive processing is facilitated.

10.3.3 Indexed Sequential File Organization

In an indexed sequential file organization, records are ordered sequentially, but can also be accessed randomly via some key.

Indexed sequential access method (ISAM) is suitable for batch processing as well as interactive processing. ISAM files are typically implemented as B-trees (or some derivative of the B-tree). It is the most widely used method of file organization (again, please review your data structures).

10.3.4 Multi-Access File Organization

In multi-access file organization, a record can be accessed by any key order. An enhancement of ISAM, multi-access file organization is typically implemented by sophisticated DBMS suites and CASE tools. The DBMS maintains the index(es) that may be defined on the file in a manner that is transparent to the user. These indexes are typically B-tree or bitmap implementations.

10.4 Summary and Concluding Remarks

Let us summarize what we have learned in this chapter.

Let us summarize what we have covered in this chapter.

- A database is the record keeping component of a software system. It must be properly designed. Failure to do so will seriously compromise the quality of the software system.
- Contemporary databases are designed to be relational or object-oriented, but mostly relational.
- Database design involves five steps: identifying the information entities (or object types), identifying relationships among the entities, eliminating unnecessary relationships, developing the ERD or ORD, and preparation of the database specification.
- There are four types of file organizations: sequential, direct, indexed-sequential and multi-access. In sequential file organization, the records are organized in arrival sequence. The file can only be accessed sequentially. In direct file organization, each record has a specific address, thereby allowing random access. Indexed-sequential file organization supports both sequential access and random access. In a multi-access file, the records can be accessed sequentially or randomly, as well as via alternate access paths.

This is merely an introduction to database design from the context of software engineering. Study of database systems is a field of computer science, so a full discussion is beyond the scope of this course (please see the recommended readings). [Appendix 10](#) provides you with a real example of the database specification for the Inventory Management System of earlier mention. The next chapter discusses design of the software user interface.

10.5 Review Questions

1. How important is database design? Cite four concrete examples of database playing an important role in computer software.
2. Identify the problems that are likely to occur due to poor

database design.

3. Identify six objectives of good database design.
4. Outline the steps involved in the design of a database.
5. The following is an excerpt from the requirements for a college academic administration system:
 - Courses are offered by various departments without any overlap (a department offers between 5 and 30 courses).
 - The courses make up academic programs, in some instances a course may occur in more than one program. Academic programs are offered by departments (no overlap allowed).
 - A faculty typically consists of several departments.
 - A lecturer is scheduled to lecture at least two courses. Each course is lectured in a specific lecture room.
 - A student may register for several courses; typically a course is pursued by a minimum of fifteen students.
 - Each student is registered to one department only.

From the information given, develop an ERD for the system.

6. The Inventory Management Information System (IMIS) of a marketing company has the following database specification:
 - The company has several warehouses, each storing certain inventory items without overlap.
 - The company has a cadre of suppliers, each supplying various items of inventory, with possible overlap.
 - The company purchases items by first sending a purchase order to a supplier (of course, the

supplier could receive several orders). Each purchase order details the items required. In responding to the purchase order, the supplier submits an invoice, detailing the items supplied, along with other relevant information.

- The company may also sell items from its inventory. In such a case, a sale-invoice is submitted to the customer, which details the items sold, along with other relevant information.
- A sale-invoice is usually with respect to a sale-order, received from a customer. A sale-order is essentially a purchase order, coming into the company, from one of its customers.
- Each inventory item belongs to a category.
- A department may make a requisition for inventory items. In response, inventory items may be issued to department(s).

From this information, develop an ERD for the system.

By conducting a brainstorming session (or otherwise), and using your E-R diagram as well as guidelines illustrated in [Figure 10-10](#), construct an initial entity specification grid (ESG) for the IMIS project.

10.6 References and/or Recommended Readings

[Chen, 1994] Chen, Peter. “The Entity-Relationship Model – Toward a Unified View of Data,” In *Readings in Database Systems* 2nd ed., pages 741-754. San Francisco, CA: Morgan Kaufmann, 1994.

[Date, 2004] Date, C. J. *An Introduction to Database Systems* 8th ed. Menlo Park, CA: Addison-Wesley, 2004. See [chapters 1, 3, 5, 6, 11, and 12](#).

[Foster, 2010] Foster, Elvis C. with Shripad Godbole. *Database Systems: A*

Pragmatic Approach. Bloomington, IN: Xlibris Publishing, 2010.

[Hoffer, 2007] Hoffer, Jeffrey A., Mary B. Prescott and Fred R. McFadden. *Modern Database Management* 8th ed. Upper Saddle River, New Jersey: Prentice Hall, 2007. See [chapters 3](#) and [4](#).

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [chapter 13](#).

[Kifer, 2005] Kifer, Michael, Arthur Bernstein and Philip M. Lewis. *Database Systems: An Application-Oriented Approach* 2nd ed. New York: Addison-Wesley, 2005. See [chapters 3](#) and [4](#).

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development With UML and Java*. Upper Saddle River, NJ: Prentice Hall, 2002. See [chapter 8](#).

[Martin, 1993] Martin, James, and James Odell. *Principles of Object-Oriented Analysis and Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1993. See [chapters 6](#) and [7](#).

[Rumbaugh, 1991] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy and William Lorenzen. *Object Oriented Modeling And Design*. Eaglewood Cliffs, NJ: Pretence Hall, 1991. See [chapter 4](#).

[Ullman, 1997] Ullman, Jeffrey and Jennifer Widom. *A First Course in Database Systems*. Upper Saddle River, NJ: Prentice Hall, 1997. See [chapters 1-3](#).

CHAPTER 11



User Interface Design

User interface management is a field of computer science that has been developed and given much attention in recent years (since the 1980's). Indeed, it is being taught as a separate subject in several curricula of Computer Science or Information Technology. Several texts have been written on the subject. It is therefore impossible to cover all that is entailed in one chapter. However, since user interface design is a very important aspect of software design, a brief summary of the subject matter is provided here.

The chapter proceeds under the following subheadings:

- Introduction
- Types of User Interfaces
- Steps in User Interface Design
- Overview of Output Design
- Output Methods versus Content and Technology
- Guidelines for Designing Output
- Overview of Input Design
- Guidelines for Designing Input
- Summary and Concluding Remarks

11.1 Introduction

A user interface is the portion of computer software that facilitates *human-*

computer interaction (HCI). Moreover, it is the window through which end users access the software system. As you are aware, most software systems have user interfaces.

User interface design is therefore applicable to all software systems that require user interaction. It is through the user interface that users communicate with the software and with each other. Many good software products have suffered neglect in the marketplace, due to poor user interface design. On the other hand, many mediocre products have managed to survive market competition, due to attractive user interface design and aggressive marketing. Proper user interface design is therefore critical, since this could determine the user acceptance and by extension, the success of the software system in the marketplace.

There are five main aspects of a user interface. These are:

- User Needs
- Human Factors
- Interface Design
- Interface Programming
- Environment

A well-designed user interface will meet the requirements in all of these areas. It will enhance effective use of the software, thus promoting end user confidence in the product. Let us briefly look at each aspect.

11.1.1 User Needs

Among the basic user needs that the user interface must address are the following:

- **Functionality** — the capacity provided the user to carry out desired tasks and activities.
- **Flexibility** — the provision of alternative approaches to solving a problem.
- **Effective Control** — users like to feel that they are in control and not the software system.

- **Reliability** — the software must offer the user some assurance that it will facilitate solution to certain problems in a consistent manner.
- **Security** — controlled access to the overall system, specific resources of the system, and data managed by the system.
- **Consistency of Design** —the user must be able to anticipate system behavior; also information must be presented to the user in a consistent manner.
- **Standardization** — the user interface must conform to established standards for the software.
- **Intelligibility** — the user interface must promote easy learning and understanding of the system.

11.1.2 Human Factors

Good user interface design must be guided by the following human factors:

- **Minimal Memory Taxation:** Taxation on human memory should be kept at a minimum.
- **Minimal Skilled Activities:** Required number of skilled tasks should be minimized; it is better to have a few skilled tasks and several operational tasks, than vice versa.
- **Shortcuts:** Shortcuts should exist for experienced users.
- **Help Facility:** There should be a help facility for all users.
- **Good Color Scheme:** The color scheme must not create pressure on the eye, but must be welcoming.
- **Minimal Assumptions:** The number of blanket assumptions about users should be kept at a minimum.

11.1.3 Design Considerations

The following considerations should be factored into the design and construction of a user interface:

- **Command Alignment:** Are the commands used appropriately aligned to typical human thinking?
- **Understandability:** How easily learned and understood are the system rules?
- **Semantic Alignment:** How aligned are the terminologies and other semantics to typical human thinking in that problem domain?
- **User Categories:** Are all user categories (experts, knowledgeable intermittent, and novices) catered for?
- **Screen Design:** How adequate are the user panels? Is there consistency among the panels?
- **System Documentation:** How adequate is the system documentation (including the help system)? Is the design appropriate?
- **System Menu:** How well-structured and adequate is the menu system?
- **User Interactions:** What kinds of user interaction are allowed?
- **System Messages:** How are feedback and diagnostic error messages handled?
- **Reversibility of Actions:** Can users reverse their actions if they need to?
- **Change Confirmation:** Do users get a chance to confirm their requests for changes before these changes take effect?
- **Locus of Control:** Who has control (or the perception of it) — the system or the user?
- **Responsiveness:** How responsive is the system?
- **Complexity Hiding:** Are users shielded from complexity details, or are they overwhelmed by the system complexity?
- **Usability:** How easy is it to learn and use the system?

11.1.4 User Interface Preparation

Preparation of the user interface involves actual development (programming) as well as preparation of the operating environment. Actual user interface development will be discussed further in [chapter 14](#). As for preparation of the operating environment, this involves various issues related to the following:

- Office preparation
- Computer site preparation

Against this background, various user interface theories and models have been developed. Your course in user interface management explores these; they will therefore not be explored any further in this chapter (for additional insights, see [Schneiderman, 2005]). Rather, we will focus our attention on user interface design alternatives.

11.2 Types of User Interfaces

User interfaces can be put into three broad categories — *menu-driven interface*, *command interface*, and *graphical user interface* (GUI). Command interface is the oldest category. Traditionally, this is how software systems were written. In order to use a command-based system, one had to first be familiarized with the command language for the system. Older operating systems such as Unix and System i (formerly OS-400) are still predominantly command-based. Next are menu-driven systems. They are characterized by menu(s) of options from which the user selects the desired option. The System i operating system is also menu-driven — you can access each system command from a menu. Many legacy systems that run on mainframes and minicomputers are menu-driven. The GUI is the newest type of user interface, and it represents the contemporary trend. You are able to use a mouse to select items (also from a menu), to drag and drop objects, and perform several other activities that we often take for granted.

[Figure 11-1](#) compares the approaches in terms of relative complexity of design (COD), response time (RT), and ease of usage (EOU) for each user interface category.

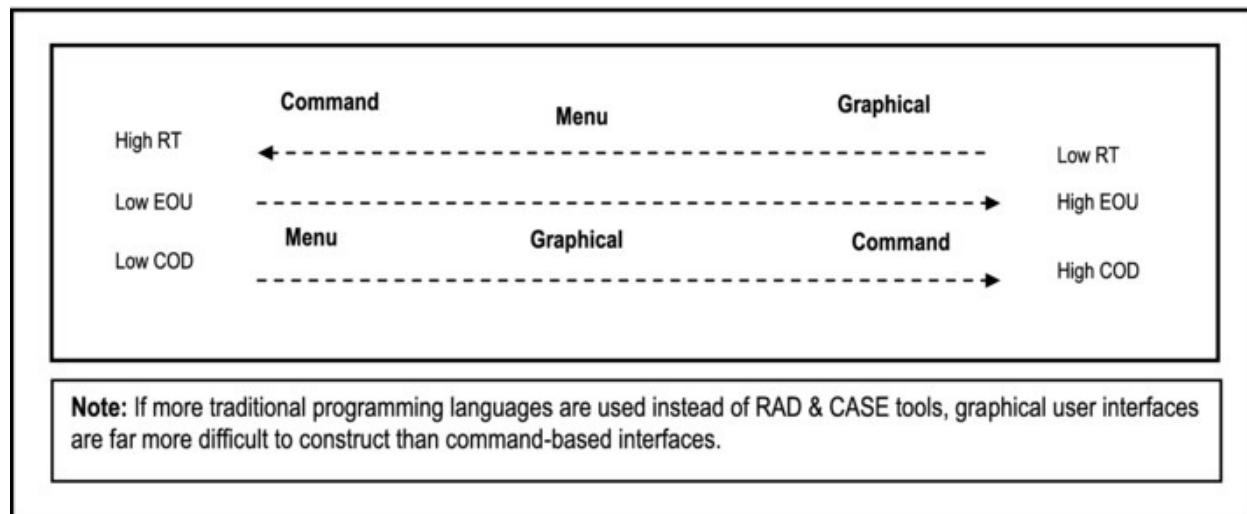


Figure 11-1. Comparison of User Interface Categories

Up until the mid-1990s, menu driven interfaces were the most frequently used, dominating the arena of business information and application systems. Since the late 1980s, GUIs have become very popular, and clearly dominate user interfaces of the current era. Of course, the approaches can be combined. An excellent example of a software system that combines all three user interface categories is the System i operating system. Being traditionally command-based, modern versions of the operating system fully support all three user interaction categories. Another example of this hybrid approach is the Windows operating system. Though predominantly GUI-based, you are allowed to key in specific system commands if you so choose. The operating systems Linux and Unix tend to be the opposite of Windows. Though predominantly command-based, each supports various GUI-based systems that can be superimposed on the underlying command-based system.

The interaction styles facilitated depend to a large extent on the type of interface supported. There are seven possible interaction styles, summarized in [Figure 11-2](#) below:

Interaction Style	Type(s) of Interface
Menu Selection	Menu, Graphical
Form Fill-ins	Menu, Graphical, Command
Command Language	Command
Natural Language	Command
Direct Manipulation	Graphical
Function Key	Menu, Command
Question - Answer	Menu, Command, Graphical

Figure 11-2. Interaction Styles for User Interfaces

11.3 Steps in User Interface Design

How you design the user interface will depend to a large extent on the type of user interface your software requires. It will also depend on the intended users of the software (experts, knowledgeable intermittent, or novices).

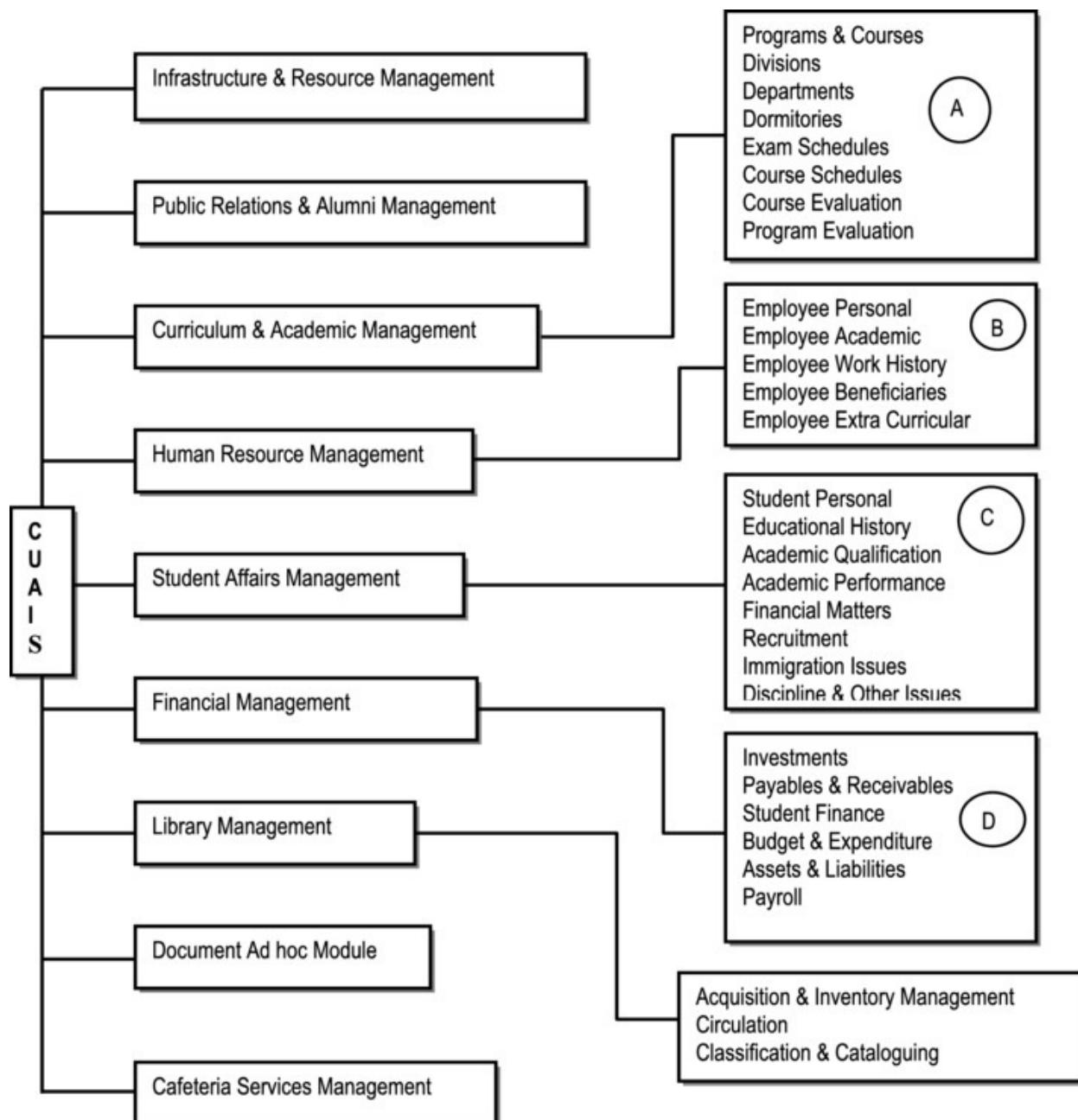
11.3.1 Menu or Graphical User Interface

If the user interface is to be menu driven or graphical, the following steps are recommended (assuming object-oriented design):

1. Put system objects (structures and operations) into logical groups. At the highest level, the menu will contain options pointing to summarized logical groups.
2. For each summarized logical group, determine the component sub-groups where applicable, until all logical groups have been identified.
3. Let each logical group represent a component menu.
4. For each menu, determine options using an object oriented strategy to structure the menu hierarchy (object first, operation last).
5. Design the menus to link the various options. Develop a

menu hierarchy tree or a user interface topology chart (UITC) as discussed in [chapter 6](#).

The partial UITC of [chapter 6](#) has been repeated in [Figure 11-3](#) for ease of reference. This chart relates to the CUAIS project of earlier discussions. Also recall from section 6.4.2 that the UITC is comparable to Schneideman's *Object-Action Interface* (OAI) model for user interfaces [Schneiderman, 2005].



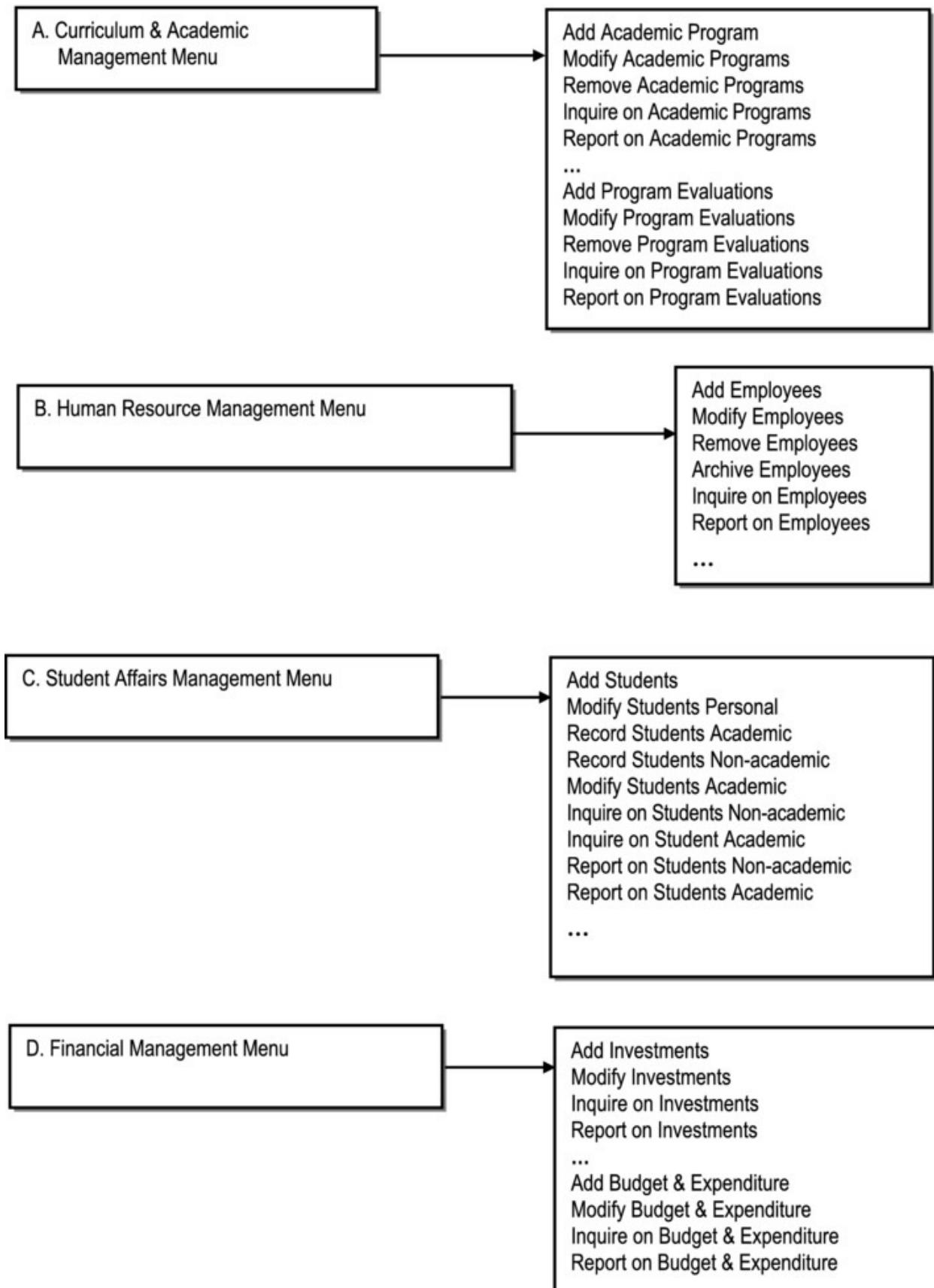


Figure 11-3. Partial UITC for the CUAIS Project

11.3.2 Command-Based User Interface

If the user interface is command driven, the following steps are recommended:

1. Develop an *operations-set* i.e. a list of operations that will be required.
2. Categorize the operations — user operations as opposed to system operations.
3. If an underlying database is involved, develop a mapping of operations with underlying database objects.
4. Determine required parameters for each operation.
5. Develop a list of commands (may be identical to operations set). If this is different from the operations set, each command must link to its corresponding system operations.
6. Define a syntax for the command language — how users will communicate with the commands (operations).
7. Develop a user interface support for each command (and by extension each operation). This interface support must be consistent with the defined command syntax.
8. Program the implementation of each operation.

Note that construction of a command-based user interface requires much more effort than a menu or graphical interface. This was not always the case: In the early days of visual programming, it was quite arduous to construct a GUI using traditional procedural programming languages. However, with the advent of object oriented CASE and RAD tools, constructing GUIs is much easier than before.

11.4 Overview of Output Design

An important aspect of the user interface is the output — the results users obtain

from the software. Output may be in the form of printed copy, VDU display, tape, diskette, CD or audio; or in the case of expert systems and CAM/CIM systems, output may be in the form of produced motion and/or products. Through output, the software communicates with the user; it is therefore an important aspect of the user interface of computer software.

Let us concentrate on the more traditional and prevalent form of output, namely information. The system must produce outputs according to user requirements. In a way, output is one fundamental test of the usefulness of a system. Some information require little processing before output; in other cases, much processing may be required before output.

Below are some important objectives of software output:

1. The output must serve the intended purpose: It must not be redundant and it must be in the required form, if it is to fulfill its desired purpose.
2. The output must fit the user and the situation: For decision support systems, it must fit individual user needs; for management information systems, it must fit functional needs; for expert systems, it must reveal expert analysis as required by the human expert; for CASE and RAD tools, it must generate accurate and accessible code; and so on.
3. The output must be in correct quantity: With the proliferation of distributed systems, this matter is not as critical as it used to be when centralized systems dominated.
4. The output must be on time: This is particularly important in mission critical systems, real time systems and traditional centralized information systems. It is not as critical in distributed system where the user decides when to generate/access system outputs.
5. The method of output (VDU, print, tape storage, disk storage, CD storage, or audio) must be appropriate to the need.

11.5 Output Methods versus Content

and Technology

The output method affects output content and presentation. For example, a screen display or a printed report will have screen heading, body and footnote; a tape or diskette file will just have raw data.

The output method also affects technology chosen to yield the output. For example, sound cards and speakers are required for audio output; printers are required for hard copy; special cameras are required for microfilm; etc.

External output leaves the organization and must adhere to standards, for such outputs. Internal output stays within the organization and must adhere to standards for such output.

In choosing output, the technological alternatives and quality factors (e.g. reliability, compatibility, and portability) are useful.

11.5.1 Printed Output

Printed output is one of the most common output methods (the other being monitor display). It is inexpensive and can serve a wide and varied users population.

For printed output, required volume also affects choices with respect to the related technology. To illustrate, the type of printer depends on the print volumes required, the speed required, frequency required, quality required.

Reports must be properly formatted, according to established standards. Usually, each operation spec (for system output) has associated output layouts for the programmer.

11.5.2 Monitor Display

Monitor display is the other most common output method. Apart from an initial cost of acquiring a visual display unit (VDU) or workstation, it is very fast and economical. Additionally, thanks to GUIs, monitor display is very attractive and effective in enhancing the user's understanding of system outputs.

Monitor display is also very convenient: the user can assess information before transferring to more expensive output medium e.g. printing.

One limitation of monitor display is that the number of users that can benefit is to some extent, constrained by the available number of monitors. Of course,

monitor display must conform to established standards.

11.5.3 Audio Output

Audio output is usually in the form of voice to a single user or multiple users in a building. This method of output has become quite common. It is useful in situations where users must be free of encumbrances, or where a message on an intercom is adequate communication to the end users.

Increasingly, individuals who have physical challenges and executives with very busy schedules are finding audio-based systems very convenient and helpful.

11.5.4 Microfilm/Microfiche

Microfilm/microfiche is traditionally used to store large volumes of data. It takes up approximately 1% of space a printed copy would take. Special machines are required to create microfilm files. Projector-like machines are then used to magnify the images so that they can be read.

Traditionally, microfilm technology has been used in legal offices, civil engineering, colleges, universities, and banks. The technology comes with a high price tag, since special equipments are required. It can also be time consuming to access information from microfilm machine.

11.5.5 Magnetic and Optical Storage

Traditionally, magnetic storage devices (disks and tapes) have been used for storing data for future usage. System backups are typically done onto disks or tapes; these devices are then stored safely until they are required. Also, before computer networks were as prevalent and sophisticated as they currently are, disks and tapes were used to transfer information between different organizations, or offices of an organization.

Optical storage devices (DVDs, CD-ROMs, CD-RWs) have become commonplace as storage media. These devices store much more information than their traditional predecessors. DVDs (digital versatile disks) are expected to dominate the future, due to their storage capacity and flexibility.

11.5.6 Choosing the Appropriate Output Method

[Figure 11-4](#) provides a comparison of the output methods. Additionally, the following factors should be considered when making decisions about system output:

1. Who will use the output?
2. How many users will access the output?
3. Where is the output required?
4. What is the purpose of the output?
5. What response speed is required?
6. What volumes are required?
7. How frequently is the output required?
8. What are the anticipated operational cost? What is the initial setup cost?
9. What are the environmental requirements?
10. How long must the output be stored?

Method	Advantages	Disadvantages
Printer	<ul style="list-style-type: none"> 1. Affordable 2. Reasonable flexibility in type, quality, location. 3. Can reach large user population inexpensively. 4. Reliable on down time. 5. Handles large volumes of output 	<ul style="list-style-type: none"> 1. May be noisy 2. May have compatibility problems with software. 3. May require special expensive supplies. 4. Requires operator intervention. 5. May be slow, depending on model.
Visual Display Unit (VDU) – also called the Monitor	<ul style="list-style-type: none"> 1. Interactive 2. Can serve indefinite no. of users, depending on number of terminals. 3. Transmission may occur over widely dispersed Network. 4. Fast, on-the-spot response. 5. Good for frequently accessed transitory information e.g. messages, mail, notices. 6. User has chance to analyze information and decide what to do with it - whether print is required. 	<ul style="list-style-type: none"> 1. User may still require printed output. 2. Can be expensive if required for many users.
Audio	<ul style="list-style-type: none"> 1. Good for individual user. 2. Good for transitory information. 3. Good if output is highly repetitive, or transmitted in an intercom or headset. 4. Ideal where worker needs to have hands free for other tasks. 	<ul style="list-style-type: none"> 1. Expensive 2. Need to ensure that output does not interfere with other activities. 3. Limited applications.
Microfilm/Microfiche	<ul style="list-style-type: none"> 1. Traditionally handles large volumes of information. 2. Reduced space required for storage. 3. Preserves fragile but frequently used materials. 4. Avoids problems of physically paging through physically cumbersome reports. 	<ul style="list-style-type: none"> 1. Requires special hardware & software. 2. May therefore be an expensive initial investment. 3. Can be effectively replaced by DVDs, CD's, Diskettes or tapes.
Magnetic/Optical Disk	<ul style="list-style-type: none"> 1. Traditionally handles large volumes of information. 2. Reduced space required for storage. 3. Avoids problems of physically paging through physically cumbersome reports. 4. Requires no special hardware different from computer system. May be cheaper than Microfilm. 	<ul style="list-style-type: none"> 1. Special hardware is required. 2. Software required to present output in a form that the user understands. 3. May therefore be more expensive than the VDU alone or printer alone.

Figure 11-4. Comparison of Output Methods

11.6 Guidelines for Designing Output

Whatever the output medium, it must follow some basic guidelines. Let us briefly focus on these guidelines as they relate to printed output and screen

output:

11.6.1 Guidelines for Designing Printed Output

The following guidelines relate to printed outputs:

1. Use information gathered during investigation phase to design reports according to user requirements.
2. Adhere to established software standards as they affect output design.
3. Use standard output design forms — headings, body and footnotes should conform to established standards.
4. Have a convention for representing variable information. Constant information is usually typed or written on output design forms. Variable information is usually indicated via some convention. For example, a string of Xs (as in XXXXXX...) is used to represent alphanumeric data; a string of 9s (as in 99999) is used to represent numeric data.
5. Decide on paper width (80, 132 or 198), quality and type.
6. Involve the user in the decision-making exercises.
7. The prototype and actual output should be well balanced and attractive.

11.6.2 Guidelines for Designing Screen Output

The above guidelines for printed output all apply to screen (monitor) output. Additionally, the following guidelines may be useful:

1. Where possible, use interactive prototypes to help the user to conceptualize the output being designed at an early stage. This may be supported by the software design tool being used (e.g. CASE tools, or presentation software with hyperlinks).

2. Keep screens simple and attractive.
3. Keep screen presentation consistent.
4. Facilitate user movement among screens.
5. Control the duplication of data on screens (duplicate only when necessary).

Screen movement is usually facilitated by one of the following:

- Scrolling
- Calling up detail (e.g. position cursor and press <Enter>)
- On-screen dialog
- Function Keys

[Figure 11-5](#) illustrates these strategies. Screen movement could also be prototyped review section 5.6) and used as a good source of feedback from prospective users of the system.

a. Scrolling:

Student Information Query/Update				
Starting Point: 2001001		Order by: - ID Number o Name		
ID Number	Last Name	First Name	Middle Name	Program
2001012	Jones	Bruce	Farnsworth	Computer Science
2001015	Barnaby	Carlos	Kane	Biology
...
2006540	McBean	Irene	Isbeth	Mathematics

F1: Exit F2: Previous Screen Scroll via the scrolling buttons or the PageUp/PageDown key
F5: Clear Search Argument PageUp: Previous Page PageDown: Next Page

b. Calling Up Detail:

Student Information Query/Update				
Starting Point: 2001001		Order by: - ID Number o Name		
ID Number	Last Name	First Name	Middle Name	Program
2001012	Jones	Bruce	Farnsworth	Computer Science
2001015	Barnaby	Carlos	Kane	Biology
...
2006540	McBean	Irene	Isbeth	Mathematics

F1: Exit F2: Previous Screen Position cursor and press <Enter> for more detail
F5: Clear Search Argument F7: Page Down F8: Page Up

c. Using Function Keys:

Student Information Query				
Starting Point: 2001001		Order by: - ID Number o Name		
ID Number	Last Name	First Name	Middle Name	Program
2001012	Jones	Bruce	Farnsworth	Computer Science
2001015	Barnaby	Carlos	Kane	Biology
...
2006540	McBean	Irene	Isbeth	Mathematics

F1: Exit F2: Previous Screen F7: Page Down F8: Page Up F5: Clear

d. Using On-Screen Dialog:

Student Information Entry	
Student ID#:	<u>930101012</u>
Name:	<u>Bruce Jones</u>
Program:	<u>B.Sc. Computer Science</u>
Lives on Hall? (Y/N)	<u> </u>

Student Information Entry	
Student ID #: 930101012	
Name:	Bruce Jones
Address Line 1:	_____
Address Line 2:	_____
State/Province:	_____
Zip Code:	_____

Note: This screen appears when the answer on the previous screen is "N"
--

Figure 11-5. Illustrating Screen Movements

11.7 Overview of Input Design

Input may be in the form of manual input, audio, magnetic and optical storage devices (output for magnetic or optical storage media is typically used as input for some other system). The arguments presented in the previous sub-section on magnetic and optical storage devices also apply here. We will therefore concentrate on manual inputs.

Desirable objectives of software inputs are as follows:

1. Input forms must capture all required data.
2. Input screens must be well-designed screens.
3. There must be well-designed forms and screens in order to

achieve effectiveness, accuracy, ease of use, consistency, simplicity, and attractiveness.

4. If input is automatic, the input files must be in the required formats.

11.8 Guidelines for Designing Input

All the points and guidelines about the design of output screens (section 11.6) are applicable here.

In the interest of clarity, they are repeated here along with some additional guidelines:

1. Use information gathered during investigation phase to design inputs according to user requirements.
2. Adhere to established software standards as they affect input design.
3. Use standard input design forms — headings, body and footnotes should conform to established standards.
4. Have a convention for representing variable information. Constant information is usually typed or written on output design forms. Variable information is usually indicated via some convention. For example, a string of Xs (e.g. XXXXXX...) to represent alphanumeric data; a string of 9s (e.g. 99999, 999.99, etc.) to represent numeric data.
5. Involve the user in the decision-making exercises.
6. The prototype and actual input should be well balanced and attractive.
7. Where possible, use interactive prototypes to help the user to conceptualize the output being designed at an early stage. This may be supported by the software design tool being used (e.g. CASE tools, or a presentation software product that supports hyperlinks).
8. Keep screens simple and attractive.

9. Keep screen presentation consistent.
10. Facilitate easy user movement among screens and reversibility of actions.
11. Control the duplication of data on screens (duplicate only when necessary).
12. As much as possible, the screen must match the associated form from which data will be entered.
13. For color monitors, avoid outrageous color schemes.
14. Input screen design must conform to established standards of the organization.
15. The principles of forms design, discussed in [chapter 2](#), must apply to the associated form. In particular:
 - ✓ The form must fulfill its intended purpose i.e. it must collect the data required.
 - ✓ It must be trivially easy to fill out the form with negligible or no error.
 - ✓ The form must be attractive.
 - ✓ The form may be designed via an appropriate software product and reviewed before implementation.

11.9 Summary and Concluding Remarks

Here is a summary of what we have covered in this chapter:

- The user interface is the window through which end users access the software system. It is critical that this component of the software system is properly designed as failure to do so could compromise the success of the software product.
- In planning the user interface, the software engineer must

dive due consideration to the end user needs, human factors, design factors, environmental factors, and actual programming of the interface.

- There are three common types of user interfaces: command interface, menu-driven interface, and graphical user interface (GUI). Your approach in designing the interface will be influenced by the type of interface that is required.
- The software engineer must observe guidelines for designing system input as well as system output. Of course, the steps taken will be constrained by the type of input/output that is required.

There is much more that could be said about user interface design, but this chapter has given you an overview that you should find useful. In fact, many computer science programs include a course in this area. Before moving on, take a look at the user interface design section of the Inventory Management System project of [appendix 10](#). The next chapter will discuss operations design.

11.10 Review Questions

1. What is a user interface? Why is user interface design important?
2. What are the five aspects of user interface design? For each of these areas, discuss the factors to be considered when a user interface is being designed.
3. Compare the three categories of user interfaces in terms of response time, ease of usage, and complexity of design.
4. Construct a grid that shows how various interaction styles apply to the different categories of user interface.
5. Outline the steps to be followed when designing a menu based user interface or a GUI. Also outline the steps to be followed when designing a command based user interface.
6. Construct a grid that compares the various output methods with respect to advantages and disadvantages of each.

7. State four basic guidelines for designing printed output.
State four basic guidelines for designing screen output.
State four basic guidelines for designing software input.

11.11 References and/or Recommended Reading

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design 6th ed.* Upper Saddle River, NJ: Prentice Hall, 2005. See [chapter 11, 12, 14](#).

[Pressman, 2005] Pressman, Roger. *Software Engineering: A Practitioner's Approach 6th ed.* Crawfordsville, IN: McGraw-Hill, 2005. See [chapter 12](#).

[Schneiderman, 2005] Schneideman, Ben. *Designing the User Interface 4th ed.* Reading, MA: Addison-Wesley, 2005.

[Sommerville, 2006] Sommerville, Ian. *Software Engineering 8th ed.* Boston, MA: Addison-Wesley, 2006. See [chapter 16](#).

[Van Vliet, 2000] Van Vliet, Hans. *Software Engineering: Principles and Practice.* New York, NY: John Wiley & Sons, 2000. See [chapter 16](#).

CHAPTER 12



Operations Design

This chapter discusses operations design as an integral part of the software design experience. The chapter proceeds under the following captions:

- Introduction
- Categorization of Operations
- Essentials of Operations Design
- Informal Methods for Specifying Operation Requirements
- Formal Specification
- Summary and Concluding Remarks

12.1 Introduction

Whether the functional approach or the object-oriented approach is employed, ultimately, software systems will necessarily have operations. An important aspect of software engineering is the preparation of *operation specifications* for the operations of a system. Hence, operations design forms a very important component of software design.

The spin-off from operations design is a set of operation specifications: each operation has an operation specification (commonly abbreviated as operation spec), which can be pulled by a programmer and used in developing the required operation (program). The more thorough the spec, the easier is the required programming effort.

In OOD, the operations are implementation of the verbs that link objects and allow communication between these objects. A common practice is to make the

operations singular (monolithic) in nature, thus further simplifying the subsequent development and maintenance processes. Further, complex operations are made to employ the services of other (simpler) operations, thus promoting code reuse and efficiency.

In the OOD paradigm, we refer to the analysis of operations (and other related issues) as *object behavior analysis* (*OBA*). You will recall that in [chapter 10](#), we used the corresponding term object structure analysis (*OSA*) to describe analysis relating to data structures of object types comprising the software system. Take some time to review the OO modeling pyramid (Figure 7.11) of [Chapter 7](#). What you need to remember is that OOD boils down to two things — *OSA* and *OBA*.

On the other hand, in FOD, the functions typify (functional) activities in the organization and facilitate management and retrieval of information; verbs may be combined in a single function. This sometimes leads to more complex functions, with a significantly lower level of reusability of code.

Example 1: For a typical information entity in a software system, the operations may be defined as follows:

In OOD for a given database entity, the legitimate operations may be: ADD, DELETE, MODIFY, RETRIEVE, QUERY, and LIST. In FOD, for the same entity, the operations (typically referred to as functions) may be MAINTAIN, QUERY, LIST, but may involve other entities.

12.2 Categorization of Operations

Categorization of the operations is a useful strategy, particularly during subsequent software development:

- Some operations will be more complex than others, and will therefore need longer development time.
- Also, by knowing about the relative complexity of operations making up a system, a project manager can make prudent decisions about work schedule.
- Some operations will be more crucial to the overall software product than others; if this information is known, important prioritization decisions can be made.

Thus, by carefully categorizing operations the software designer can significantly contribute to the success of subsequent software development and maintenance. [Figure 12-1](#) provides a four-step approach to categorizing operations.

1. Rank the major quality factors (mentioned in chapters 1 and 3) based on requirements of the software.
2. Rank the operation, based on the quality factors ranking.
3. Prioritize the operations as mandatory, important, or optional but desirable (i.e. nice to have).
4. Apply a *relative complexity index* to each operation. This may be done mathematically by considering factors such as the number of other system objects (including database objects) that the operation has to communicate with, the anticipated nature of these interactions, the nature of algorithms to be employed, and the anticipated code length. However, in practice, this is usually estimated by an experienced software engineer, after considering these factors.

[Figure 12-1. Categorizing Operations](#)

While the first two steps of [Figure 12-1](#) may be considered optional, the latter two are essential, as they help in guiding decisions about the project during the development phase.

12.3 Essentials of Operations Design

Each operation must be assigned a unique system (implementation) name. Preferably, these system names should conform to the established naming convention that is in vogue (review [chapter 9](#)).

For each operation, the following should be clearly stated: what the operation will do; the inputs and outputs; the algorithm(s) to be implemented; any required validation rules and/or calculation rules. These things are normally provided in an *operation specification* (often abbreviated as operation spec or op spec).

In the interest of clarity and efficiency, operation specs are usually expressed in a standard format, according to the software development standards of the organization (review [chapter 9](#)). Over the years, several formal methods as well as informal methods for software specification have been proposed. The next two sections will summarize some of the commonly used approaches, as well as a pragmatic approach, introduced by the author.

As you proceed, bear in mind that an operation will be implemented by code in a particular programming language. How it is implemented depends on the development environment: In an FO environment, it is typically implemented as a program. In a purely OO environment, the operations may be implemented as a

method of a class, a class consisting of several related methods, or a set of related classes. This is also the likely case in a hybrid environment (consisting of an OO user interface superimposed on a relational or object database).

12.4 Informal Methods for Specifying Operation Requirements

Remember, in operations specification, the software engineer is attempting to define and express the requirements of system operations in a manner that promotes comprehension and efficiency, while avoiding the hazard of being ambiguous or too verbose.

In this section, we shall briefly review some traditional approaches to operations specification. We will then look at use of the Warnier-Orr diagram and the UML notation. The section closes with a discussion of a methodology introduced by the author.

12.4.1 Traditional Methods

Traditional informal methods of operations specification include program flowcharting, the use of IPO charts, decision techniques, and pseudo-coding (review [chapter 5](#)). The main advantages of these approaches are:

- Visual aid (in the case of IPO chart and program flow chart)
- Flexibility and creativity in treating unanticipated or complex situations

However, these approaches have inherent flaws, some of which are stated below:

- None of these approaches lends itself to comprehensive coverage of all the requirements of operations of a software product. To illustrate, a flowchart does not readily provide information about the inputs to and outputs from an operation as well as an IPO chart. An IPO chart does not represent logical decisions as well as a flowchart or decision

table. Neither does a pseudo-code.

- Pseudo-codes and flowcharts can be difficult to maintain, especially for a large, complex system.
- The use of pseudo-code does not facilitate easy analysis; it does not avoid the possibility for ambiguity, since there is no established standard vocabulary; neither does it facilitate easy automation.

In view of the foregoing, the IPO chart of [Figure 12-2](#) should illustrate the limitations of the technique. The figure shows an IPO chart for an operation that facilitates addition, modification, or deletion of employee records in a human resource management system (HRMS), or some other system requiring employee information. Traditionally, operations that provide such functionalities (addition, update, and deletion) were called MAINTAIN operations, and appeared frequently in software systems designed in the FO paradigm.

Operation Name: HREmployee_MO		
Operation Description: Allows maintenance of the employee personal information.		
Input	Processing	Output
HREmployee_BR	1. Accept employee number	Edit List, HREmployee_BR
Employee Information Form	2. If this is a new employee number, allow addition of a new employee	
	3. If this is a preexisting employee number, find out if the user desires modification or deletion	
	4. If modification is chosen, allow modification of the employee record; otherwise, allow deletion of the employee record	

[Figure 12-2](#). IPO Chart for a MAINTAIN Operation

12.4.2 Warnier Orr Diagram

In the Warnier-Orr approach, the operation is consistently broken down into constituent activities, in a hierarchical manner. Component activities are numbered in a manner similar to the sections of the chapters of this book. The basic idea is as shown in [Figure 12-3](#).

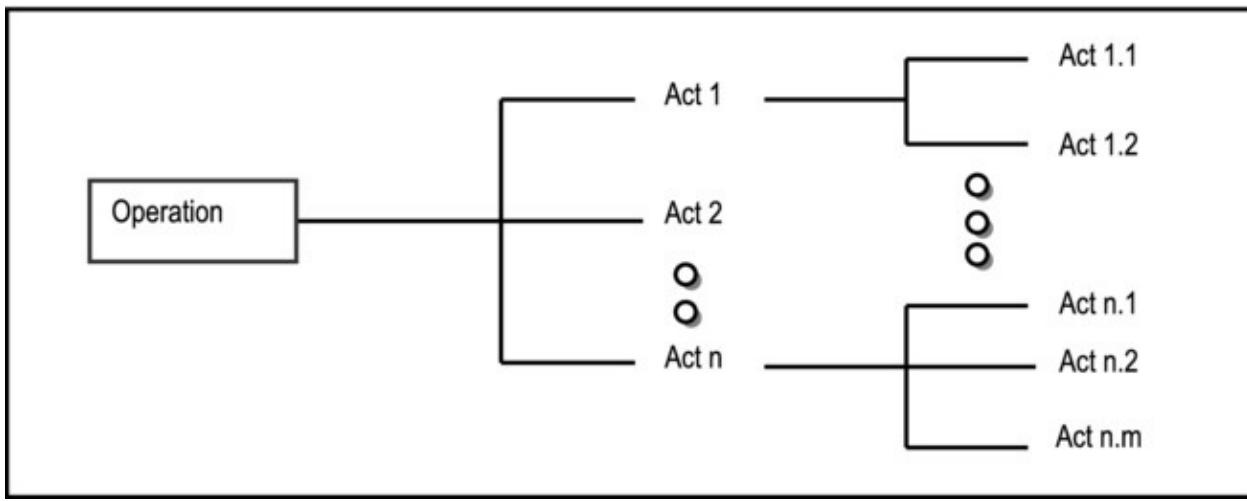


Figure 12-3. Illustration the Warnier-Orr Method

A Warnier-Orr diagram provides two main advantages:

- It shows the relationships among activities of an operation.
- If activities are logically arranged it can replace the flowchart/pseudo-code.

Disadvantages of the Warnier-Orr Diagram Are

- It does not explicitly show inputs to or outputs from an operation.
- It does not include explicit representation of logical decisions.
- The diagram can grow indefinitely, so that in the absence of a CASE tool that supports the technique, maintaining Warnier-Orr diagrams for system operations could be quite time consuming.

Although the technique might appear to have a function-oriented connotation, it is in fact also applicable in an object-oriented scenario. [Figure 12-4](#) provides an example of a Warnier-Orr diagram for a MAINTAIN operation for employee records. The operation name (**HREmployee_MO**) is indicated in the first activity box. Still referring to the figure, the item **HREmployee_BR** may be implemented as an object type in an OO environment, or as a relational table in a relational database that is access by an appropriately constructed user interface

(which may very well be based on an OO software development environment). This operation would form part of the user interface.

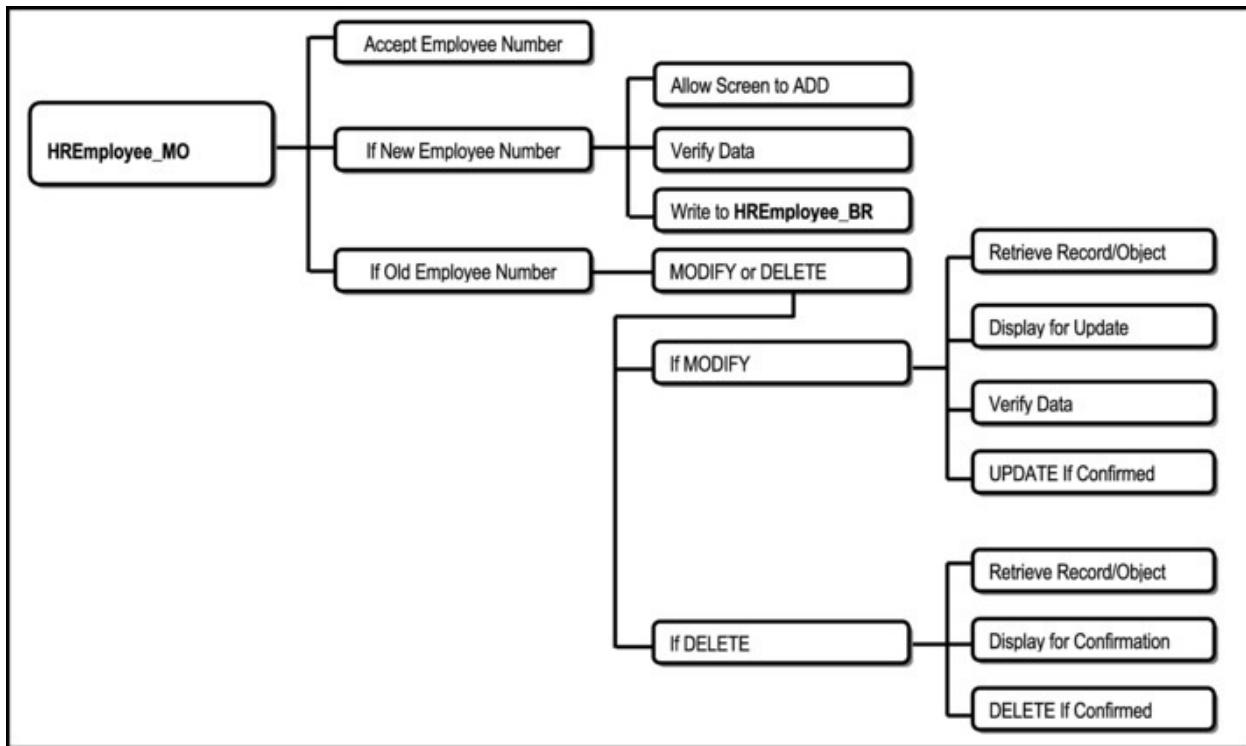


Figure 12-4. Warnier-Orr Diagram for a MAINTAIN Operation

12.4.3 UML Notations for Object Behavior

The unified modeling language (UML) employs a comprehensive set of notations for specifying the behavior of objects when using the object-oriented approach to software design. In conducting OBA, the UML facilitates the following techniques:

- Use-case Diagrams
- State Diagrams
- Activity Diagrams
- Sequence Diagrams and Collaboration Diagrams

State diagrams (also called finite state machines) were discussed in [chapter 6](#). A full discussion of the other techniques is best deferred for your course in

OOM. However, [appendix 6](#) provides an overview. Please review [figures 10-10](#) and [10-11 of chapter 10](#). In those figures, some of the information entities that would be found in the CUAIS project (for a generic college or university) were represented. One such entity (object type) was the **Employee**. The following figures ([Figures 12-5 – 12-8](#)) provide examples of a use-case diagram or a typical **Employee** object, a state diagram for a typical **Employee** object, (copied from [chapter 6](#)), an activity diagram for creating (i.e. adding) an **Employee** object, and a collaboration diagram for querying (i.e. inquiring on) an **Employee** object.

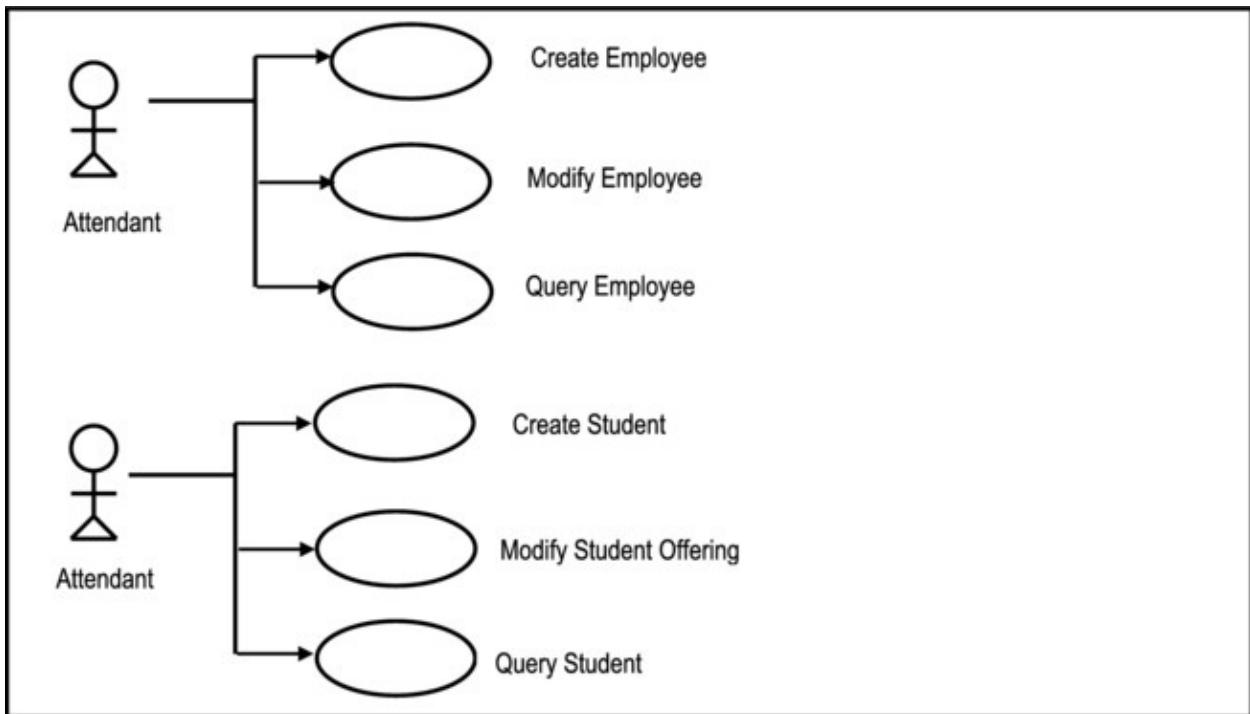


Figure 12-5. Use-case for Employee Processing and Student Processing

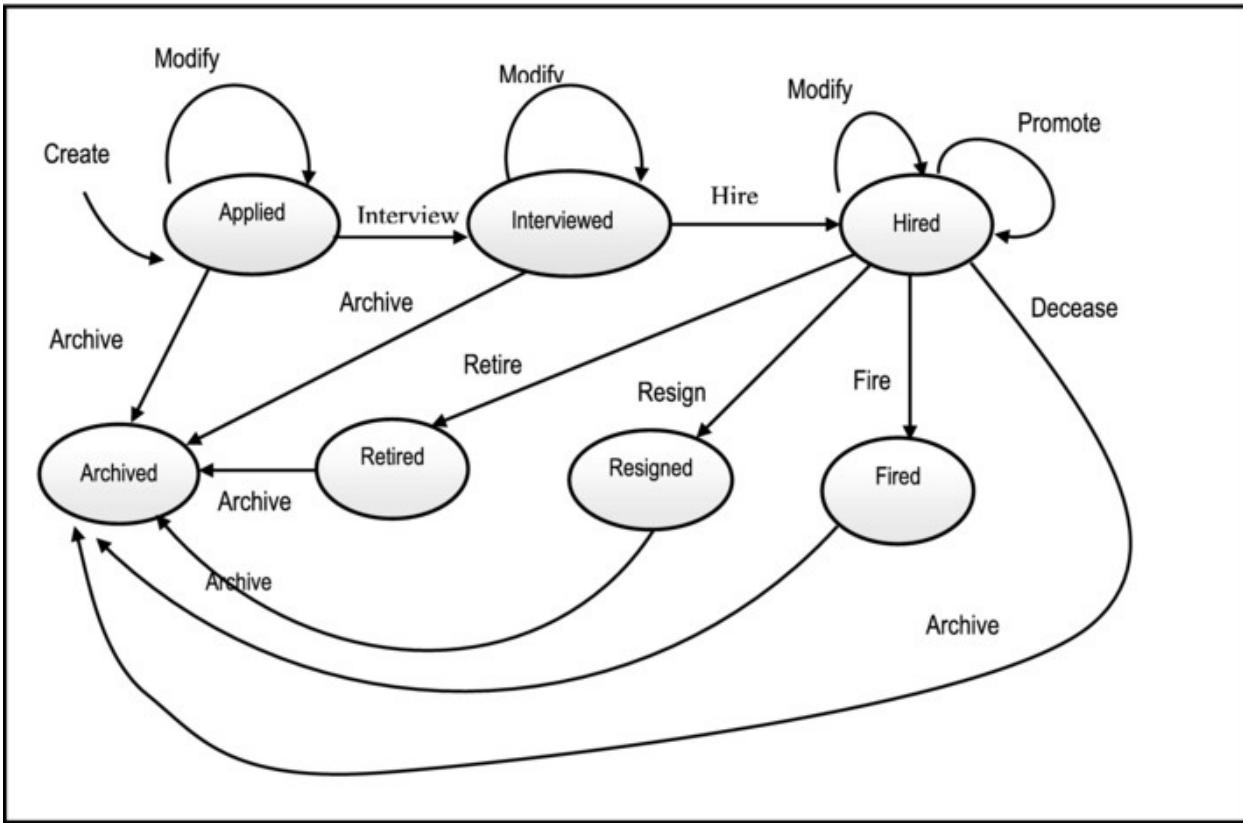


Figure 12-6. State Diagram for Employee Object

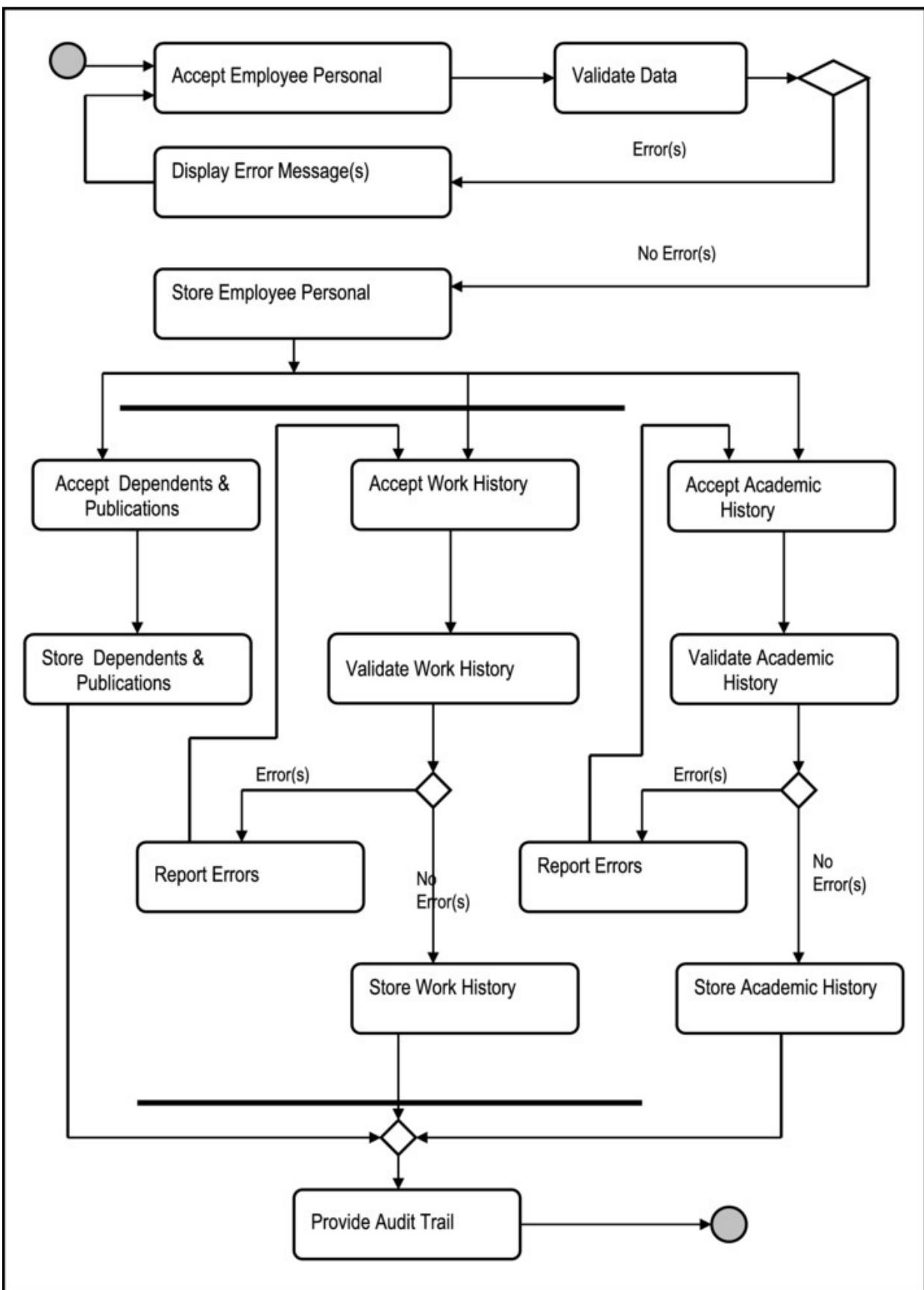


Figure 12-7. Activity Diagram for Adding Employee Information

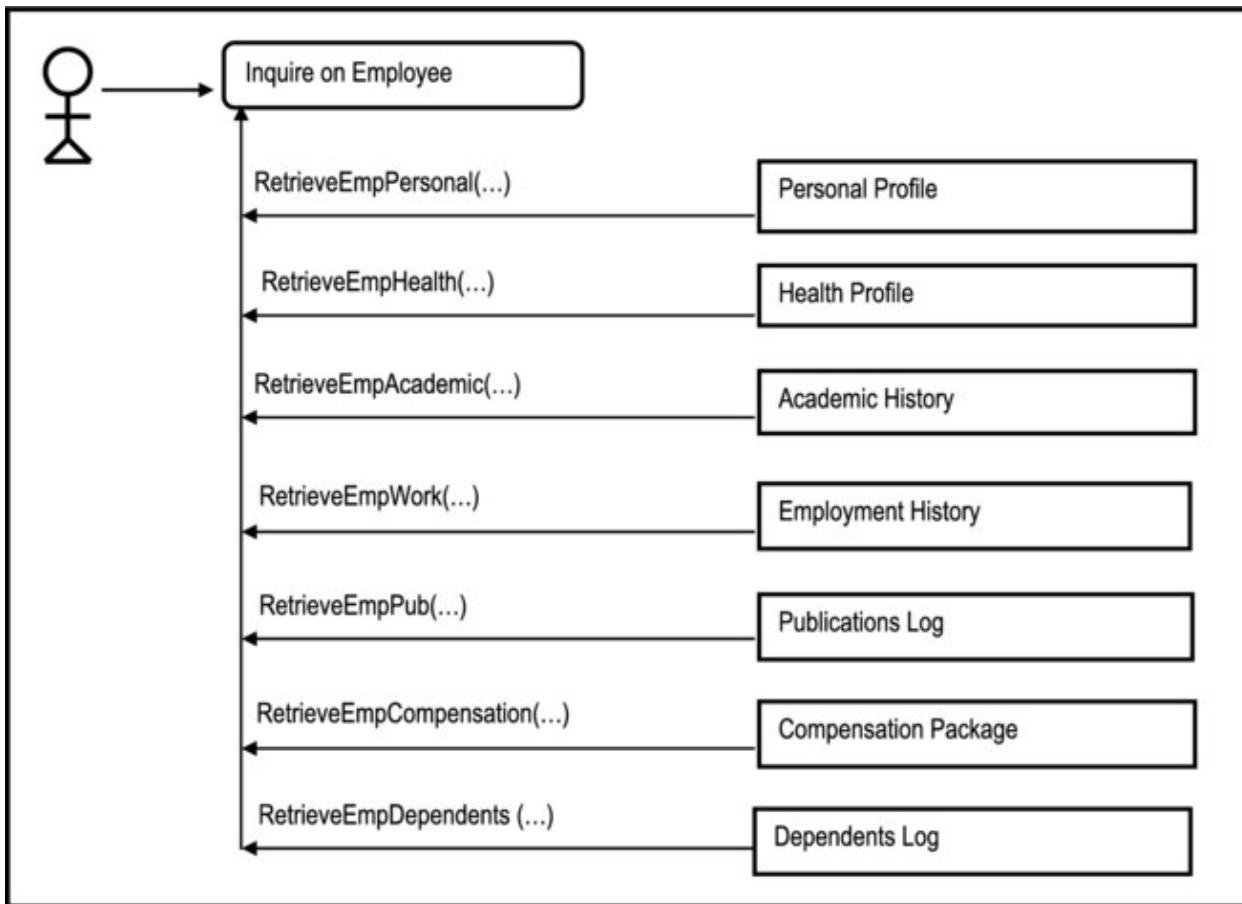


Figure 12-8. Collaboration Diagram for Inquiry on Employee Information

12.4.4 Extended Operation Specification

The *extended operation specification* (EOS) approach was developed by the current writer, and has been successfully used on various systems designed by him. In this approach the software engineer records important requirements about an operation in such a manner as to offset the disadvantages of the previously discussed methods.

The basic idea is to provide enough detail about the required operation, so that a programmer that pulls the spec should have little or no problem in writing the operation (program). The format recommended is shown in [Figure 12-9](#). Observe that the specification also includes categorization information, which may be used during software development. The technique is applicable to both

FOD and OOD.

System:	
Sub-system:	
Operation Name:	
Operation Description:	
Operation Categorization: [] (Mandatory, Important, Nice)	Complexity Rank [] of Total Possible Rating []
Spec. Author: _____	Spec. Date: ___/___/___
Inputs:	_____
	...

Outputs:	_____
	...

Validation Rules:	_____
	...

Special Notes :	_____
	...

Outline:	_____
	...

Figure 12-9. Components of the Extended Operation Specification

Note:

1. The required output formats and/or screen formats could be (and are usually) designed and attached.
2. Validation rules are itemized and special notes are itemized.
3. The outline could be in the form of a pseudo-code (as illustrated in the [Figures 12-11 – 12-14](#)), a flowcharts, a Warnier-Orr diagram, an IPO Chart, or an activity

diagram.

[Figure 12-10](#) provides an excerpt from the object/entity specification grid (O/ESG) of [Chapter 10](#), repeated here for convenience; here, we shall concentrate on the **Employee** object type (entity). [Figures 12-11 – 12-14](#) provide sample EOSs for an ADD operation, an UPDATE operation, a DELETE operation and an INQUIRE operation respectively.

E2 – Employee [HREmployee_BR]	
Attributes:	
1. Employee Identification Number [EmpNo] [N7]	
2. Employee Last Name [EmpLName] [A20]	
3. Employee First Name [EmpFName] [A20]	
4. Employee Middle Initials [EmpMInitl] [A4]	
5. Employee Date of Birth [EmpDOB] [N8]	
6. Employee's Department [EmpDepNo] [N4] {Refers to E1}	
7. Employee Gender [EmpGender] [A1]	
8. Employee Marital Status [EmpMStatus] [A1]	
9. Employee Social Security Number [EmpSSN] [N10]	
10. Employee Classification Code [EmpClass] [A3] {Refers to E3}	
....	
Comments:	
This table stores standard information about all employees in the organization.	
Indexes:	
1. Primary Key: [1] – constraint HREmployee_PK	
2. HREmployee_NX1 on [2, 3, 4]	
3. HREmployee_NX2 on [7]	
Valid Operations:	
1. Manage Employees [KREmployee_MO]	
1.1. Add Employees [HREmployee_AO]	
1.2. Update Employees [HREmployee_UO]	
1.3. Delete Employees [HREmployee_ZO]	
2. Inquire on Employees [HREmployee_IO]	

[Figure 12-10](#). Sample Object/Entity Specification Grid for Employee

Operation Biography:

System: CUAIS
Subsystem: Human Resource Management
Operation Name: **HREmployee_AO**
Operation Description: Facilitates addition of records to the employee file.
Operation Category: Mandatory Complexity Rank: 6 of 10
Spec. Author: E. Foster Date: 24-10-94

Inputs:

Employee Personal Information — **HREmployee_BR**
Department — **HRDepartment_BR**
Employee Classification — **HRClassif_BR**
New Employee Profile Form

Outputs:

Employee Personal Information — **HREmployee_BR**
Audit Trail (edit listing)

Validation Rules:

1. Employee # must not previously exist.
2. Employee classification code must exist in file **HRClassif_BR**.
3. Employee's Department Number must exist in file **HRDepartment_BR**.
4. Blank or null name and address must be rejected.
5. Date of birth must be a valid date in the 20th century.

Special notes: None

Operation Outline (Pseudo-code):

```
START
WHILE (User wishes to continue)
    Accept Key Field(s);
    Check Record Absence or Existence in file HREmployee_BR;
    IF      (Record Absent)
        Accept Non-key Fields;
        Validate Non-key Fields based on Validation Rules;
        WHILE (Any Error Exists),
            Re-display Non-key Fields for possible Update;
            Display appropriate error message(s);
            Validate Non-key Fields based on Validation Rules;
        END WHILE;
        Re-display full Record for confirmation;
        IF      (Confirmation Obtained)
            Write New Record to file HREmployee_BR;
            Write New Record to audit file for Additions;
        ENDIF;
        ELSE   Inform the User that nothing was saved; END-ELSE;
    ENDIF;
    ELSE   Display Message ('Record already exists'); END-ELSE;
    Check if User wishes to quit and set an exit flag if necessary;
END WHILE;
Generate Edit-List;
STOP
```

Figure 12-11. Sample EOS for ADD Operation

Operation Biography:

System: CUAIS
Subsystem: Human Resource Management
Operation Name: **HREmployee_UO**
Operation Description: Facilitates update of employee records in the employee file.
Operation Category: Mandatory Complexity Rank: 6 of 10
Spec. Author: E. Foster Date: 24-10-2004

Inputs:

Employee Personal Information — **HREmployee_BR**
Department — **HRDepartment_BR**
Employee Classification — **HRClassif_BR**
New Employee Profile Form

Outputs:

Employee Personal Information — **HREmployee_BR**
Audit Trail (edit listing)

Validation Rules:

1. Employee # must previously exist.
2. Employee classification code must exist in file **HRClassif_BR**.
3. Employee's Department Number must exist in file **HRDepartment_BR**.
4. Blank or null name and address must be rejected.
5. Date of birth must be a valid date in the 20th century.

Special notes: None

Operation Outline (Pseudo-code):

```
START
WHILE (User wishes to continue)
    Accept Key Field(s);
    Check Record Absence or Existence in file HREmployee_BR;
    IF      (Record Present)
        Retrieve Record and update Audit Log Fields (with before-values);
        Display Non-key Fields for possible Update;
        Validate Non-key Fields based on Validation Rules;
        WHILE (Any Error Exists),
            Re-display Non-key Fields for possible Update;
            Display appropriate error message(s);
            Validate Non-key Fields based on Validation Rules;
        END-WHILE;
        Re-display full Record for confirmation;
        IF      (Confirmation Obtained)
            Update Audit Log Fields (with current-values);
            Write New Record to audit file for Updates;
            Update Record in file HREmployee_BR;
        ENDIF;
        ELSE Inform the User that nothing was saved; END-ELSE;
    ENDIF;
    ELSE Display Message ('Record does not exists'); END-ELSE;
    Check if User wishes to quit and set an exit flag if necessary;
END-WHILE;
Generate Edit-List;
STOP
```

Figure 12-12. Sample EOS for UPDATE Operation

Operation Biography:

System: CUAIS
Subsystem: Human Resource Management
Operation Name: **HREMPLE_ZO**
Operation Description: Facilitates deletion of employee records from the employee file.
Operation Category: Mandatory Complexity Rank: 6 of 10
Spec. Author: E. Foster Date: 24-10-2004

Inputs:

Employee Personal Information — **HREmployee_BR**
Department — **HRDepartment_BR**
Employee Classification — **HRClassif_BR**
New Employee Profile Form

Outputs:

Employee Personal Information — **HREmployee_BR**
Audit Trail (edit listing)

Validation Rules:

1. Employee # must previously exist.

Special notes: None

Operation Outline (Pseudo-code):

```
START
WHILE (User wishes to continue)
    Accept Key Field(s);
    Check Record Absence or Existence in file HREmployee_BR;
    IF      (Record Present)
        Retrieve Record;
        Display full Record for confirmation;
        IF      (Deletion Confirmation Obtained)
            Update Audit Log Fields (with current-values);
            Write New Record to audit file for Deletions;
            Delete Record from file HREmployee_BR;
        ENDIF;
        ELSE   Inform the User that nothing was saved; END-ELSE;
    ENDIF;
    ELSE   Display Message ('Record does not exists'); END-ELSE;
    Check if User wishes to quit and set an exit flag if necessary;
END WHILE;
Generate Edit-List;
STOP
```

Figure 12-13. Sample EOS for DELETE Operation

Operation Biography:

System: CUAIS
Subsystem: Human Resource Management
Operation Name: **HREmployee_IO**
Operation Description: Facilitates inquiry on employee information.
Operation Category: Mandatory Complexity Rank: 9 of 10
Spec. Author: E. Foster Date: 24-10-2004

Inputs:

Employee Personal Information — **HREmployee_BR**
Department — **HRDepartment_BR**
Employee Classification — **HRClassif_BR**

Outputs: Monitor Display

Validation Rules: None

Special notes:

1. It will be possible to query employee information via any of the following access paths:
 - 1.1 By Identification Number or Social Security Number
 - 1.2 By Name
 - 1.3 By Department, Classification and Name
 - 1.4 By Department, Gender and Name
 - 1.5 By Department and Name
 - 1.6 By Classification, Gender and Name
 - 1.7 By Classification and Name
2. Each option will invoke one of seven sub-operations (HREmployee_IO1, HREmployee_IO2, ... HREmployee_IO7).
3. Utilizes the logical view HREmployee_LV1 which joins HREmployee_BR, HRDepartment_BR and HRClassifBR.

Operation Outline:

START: /* Update */

While User Wishes to Continue

 Present the User With the Options mentioned above;

 Depending on the User's Choice, Invoke one of sub-operations HREmployee_IO1, ...

 HREmployee_IO7;

 End-While

STOP.

/* Each sub-operation will allow the user to specify appropriate search criteria; this will then be used to retrieve records from the system and display them on the monitor. For instance, */

...

Outline for HREmployee_IO5:

START

While User Wishes to Continue

 Prompt user for Starting Department and Name;

 Starting at that point in **HREmployee_BR**, Load and Display a *Virtual Data Collection Object* with all records until End-of-File;

End-While;

STOP.

Figure 12-14. Sample EOS for INQUIRE Operation

Three additional points to note from the illustrations ([Figures 12-11 – 12-14](#)):

1. The algorithms for adding, modifying and deleting data items (records) have been standardized and can be used for different scenarios. How they are implemented will vary for different software development tools.
2. The term *Virtual Data Collection Object*, as used in [Figure 12-14](#), refers to any user interface widget that may typically be employed by a RAD or CASE tool (for instance, in Delphi, we could use a DB-Grid or a DB-Image; in Team Developer, we could use a Child-Table).
3. The term *logical view* has also been introduced in [Figure 12-14](#). A logical view is a virtual database object which stores an access path to data contained in persistent database tables. In the example, we have a join logical view, based on the fact that there is a relationship between **E2 (HREmployee_BR)** and **E1 (HRDepartment_BR)** on the one hand, and **E2 (HREmployee_BR)** and **E3 (HRClassif_BR)** on the other (see [Figure 12-10](#) and review [Figure 10-9 of Chapter 10](#)). You will spend much more time working with logical views in your database systems course.

Among the advantages of EOS are the following:

- It allows the software engineer to pack all the relevant information about an operation into one spec so that development is easy.
- It provides information that allows the project manager to make intelligent work assignments during software development.
- Under the operation outline section, the software engineer has the flexibility of using a program flowchart, a pseudo-code, a Warnier-Orr diagram, or an activity diagram.
- Important information such as I/O requirements, categorizations, etc. can be included in the spec.

- The whole process of specifying an EOS for an operation can be automated by developing a software system for that purpose.

In terms of disadvantages of the EOS, no major drawback of the methodology has been identified so far. However, with time it is anticipated that constructive criticisms will inspire further refinement.

12.5 Formal Specifications

A formal specification of software is a specification that is expressed in a language whose syntax and semantics have been formally defined.

Formal specification methods have been developed and are widely used in software engineering. Several *program description languages* (PDLs) have been proposed and used in software engineering. Some examples are mentioned below:

- PSL — Problem Statement Language
- Ada PDL
- Z-specifications (pronounced zed specifications)
- Larch Specifications
- B Specifications
- Lotos Specifications

The prediction that by the twenty first century, a significant proportion of software systems would be developed by formal methods has not been realized, due to a number of reasons:

1. Successful informal methods such as structured methodologies and OO methodologies have been on the increase.
2. Market dynamics puts pressure on the software engineering industry to produce software at much faster rates than formal methods would allow.

3. Formal methods are not well suited for some scenarios, e.g. user interface development.
4. Formal methods provide limited scope for quality factors such as scalability and portability. To some extent, maintainability and flexibility are also negatively affected. As you are aware, these are crucial requirements for contemporary software.

One significant advantage of formal methods is that they force precise, unambiguous specification of software. Because of this, formal methods are widely used in areas of software engineering where precision is required. Two examples of such areas are hardware synthesis and compilation. However, to do justice to the field, further exploration of formal specifications is best treated in a course on formal methods.

12.6 Summary and Concluding Remarks

It's time once more for us to summarize what we have covered in this chapter:

- Operations design is an integral part of the OBA process (assuming the OOD paradigm). The spin-off from operations design is a set of operation specifications: each operation has an operation specification that outlines the blueprint for the operation.
- Categorizing operations is very useful particularly during the development of the software system. Each operating can be categorized by giving consideration to its alignment with quality factors or importance to the software system, the level of importance of the operation, and its relative complexity.
- Each operation spec must have a unique name, followed by unambiguous guidelines that will help a programmer to easily write the actual operation.

- Informal methods of operations specifications include (but are not confined to) the following: traditional methods (program flow charts, pseudo-code, and IPO charts); Warnier-Orr diagrams; UML diagrams (use-case diagrams, state diagrams, activity diagrams, sequence diagrams, and collaboration diagrams); EOS formulations.
- Formal methods of operations specifications include (but are not confined to) PSL, ADA/PDL, Z-specifications, Larch specifications, B specifications and Lotos specifications.

Armed with the software development standards, architectural specification, the database specification, the user interface specification, and the operation specification, you are almost ready to embark on the actual software development with confidence. Bear in mind that if you are using an OO-CASE tool, actual design and development may be merged into one modeling phase, since many of the diagrams may be executable diagrams. [Appendix 6](#) provides additional discussion and illustrations on OBA, and [appendix 10](#) provides several examples of operation specifications (from the Inventory Management System of earlier mention). We still have a few design issues to cover, and these will be discussed in the next chapter.

12.7 Review Questions

1. How important is operations design? Explain how it affects the development of a software system.
2. Outline an approach for categorizing operations comprising a software system.
3. The **Student** entity would be an important component of the CUAIS project. It contains attributes **StudentID**, **Name**, **Gender**, **DateOfBirth**, **Major**, **Dept#**, among others. Each **Student** object has a unique identification number. The department (**Dept#**) to which a student is assigned must previously exist in the **Department** entity. Also, the student's major must reside in the **AcademicProgram** entity. **Gender** must be male or female and **DateOfBirth** must be a valid date in the 20th

or 21st century.

- 3a. Propose an O/ESG for the **Student** entity.
- 3b. Propose an EOS for the operation to allow addition of valid student records. Your outline may include a Warnier-Orr diagram, a pseudo code, or an activity diagram.
- 3c. Propose an EOS for an operation to allow users to run interactive query of student information.
- 3d. Propose a collaboration diagram for interactive query of student information.
4. Examine [Figure 12-6](#) and thoroughly explain all the state transitions and the operations that trigger them.
5. Examine [Figure 12-7](#) and thoroughly explain the behavior of the **ADD Employee** operation.
6. Examine [Figure 12-8](#) and thoroughly explain it.
7. When is formal specification relevant? Give three examples of formal specification languages.

12.8 References and/or Recommended Reading

[Peters, 2000] Peters, James F. and Witold Pedrycz. *Software Engineering: An Engineering Approach*. New York, NY: John Wiley & Sons, 2000. See [Chapter 5](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice 3rd ed.* Upper Saddle River, NJ: Prentice Hall, 2006. See [Chapter 4](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering 6th ed.* Boston, MA: McGraw-Hill, 2005. See [Chapter 16](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Boston, MA: Addison-Wesley, 2006. See [Chapter 10](#).

[Van Vliet, 2000] Van Vliet, Hans. *Software Engineering: Principles and Practice*. New York, NY: John Wiley & Sons, 2000. See [Chapter 15](#).

CHAPTER 13



Other Design Considerations

We have looked at all the major aspects of software design. However, there are a few outstanding areas that need some attention. This chapter covers these areas as outlined below:

- System Catalog
- Product Documentation
- User Message Management
- Design for Real-Time Software
- Design for Reuse
- System Security
- Summary and Concluding Remarks

13.1 The System Catalog

The system catalog (also called data dictionary) is a very useful and effective management and analysis tool for the software engineer or system manager. It stores critical control information that will be useful to anyone managing or working on the development/maintenance of the software.

Typically it commences its existence during the design phase, but may be introduced as early as during the preparation of the requirements specification. Once introduced, it serves the project for the rest of its useful life.

13.1.1 Contents of the System Catalog

The system catalog contains database related information as well as operational control information.

Essential database-related contents of the system catalog are the following:

- Name, description, proper coding and aliases of each information entity (or object type) in the system
- Name, description, proper characteristics (e.g. type and length) and coding of data element in each information entity
- Required editing and integrity checks on data elements
- Indication as to whether a data element references another element in another entity (object type) and the type of reference (relationship)

Operational contents include:

- Name and description of sub-systems (and/or modules)
- Name and description of each operation (or function) — including super-operations as well as sub-operations
- For each operation, specification of entities (object types) used and/or impacted

Business rules include:

- Relationships among entities (object types)
- Data integrity rules (if not already specified above)
- Calculation rules for operations
- Other operational rules

Note In purely OO environments, the approach is somewhat different: For each object type, a set of allowable operations is specified. The reason for this is that purely OO software tools will allow the software to encapsulate the object's related operations with its data structure into a class (review [chapter 9](#)).

13.1.2 Building the System Catalog

The system catalog may be constructed in any of three ways:

Via Static Tables: In the absence of a sophisticated software development environment, the catalog can be built using static table(s) via some word processor e.g. Word Perfect, Microsoft Works, Microsoft Word etc. Maintenance will be particularly challenging, since any change to the model will necessitate manual changes to the tables.

Via Dynamic Files: Again assuming an environment devoid of sophisticated software development tools, an alternative to static tables would be to create data file(s) that can store the pertinent information, then load and manage the information via some entry/update application program(s). This could be done in any programming language.

Automatic Creation: A few business-oriented operating systems (e.g. System i) have the facility to allow the software engineer to define and maintain a data dictionary as an important component of software developed in that environment. Additionally, the more sophisticated software development products (CASE tools and DBM suites e.g. Oracle, DB2, Informix, etc.) have built-in features to automatically build and maintain a system catalog, while a software product is being developed. The software engineer can then access this catalog.

Whichever of the first two methods is employed, a top-down approach to defining the system catalog is recommended:

- Specify all sub-systems of the software system
- Specify all entities (object types)
- Specify all elements of entities
- Specify all operations

13.1.3 Using the System Catalog

The system catalog is an excellent means of system documentation:

- You can obtain assorted views of the system e.g. entity-operations lists; operation-entities lists; entity-elements lists; terms and aliases; list of system operations; list of system entities; etc.
- You can also obtain information about relationships. In the case of automatic creation, you might even be able to obtain an ERD/ORD from the catalog.
- You will also be able to obtain assorted views of the systems business rules on a system-wide basis, by operation, or by entity (object type).

The system catalog is very effective in guiding the design phase (particularly with respect to database, operations, menu interface, and business rules). It can also be used as a management tool in design, development and maintenance phases. [Figure 13-1](#) provides a very basic illustration of the kind of information that a system catalog could provide.

Information Entities / Object Types					
Entity/OT Name	Descriptive Name	Subsystem/Module	Comment		
AMProgram_BR	Academic Program	Academic Management	Stores definition of all academic programs		
AMCourse_BR	Course	Academic Management	Stores definition of all courses offered		
...					

Operations					
Operation Name	Operation Title	Subsystem/Module	Inputs	Outputs	Comment
AMProgram_AO	Add Academic Program	Academic Management	Keyboard Entry	AMProgram_BR	Allows addition of academic programs
...					

Operational Business Rules					
Operation Name	Rule(s)				
AMProgram_AO	1. Program Code must not previously exist on file.				
	2. Program Title is mandatory.				
	3. Program must be offered by an existing academic department.				
...					

Database Business Rules (Relationships)					
Entity/OT Name	Referenced Entity / Object Type	Ref-Type	Comment		
AMProgStruct_BR	AMProgram_BR	M:1			
	AMCourse_BR	M:1			
...					

Note:

1. The entity names and operation names used here are based on the naming convention described in appendix 10.
2. The catalog can also be used to provide information on the properties (elements) comprising each entity (or object type).

Figure 13-1. Illustration of System Catalog

13.2 Product Documentation

Product documentation typically involves three main areas: the *system help facility*, the *users' guide*, and a more technical document often referred to as *the system guide*. Of course, there are variations according to the product, as well as the organization of responsibility. For instance, the users' guide and system guide may be merged as one document or set of documents. Also, if the software system is very large and complex, there may be a *product overview* document.

13.2.1 The System Help Facility

In help design, the software engineer specifies exactly how the help system will

be structured and managed for the software in question. Two important considerations here are:

- The structure of the help system
- Content of the help system

Structure of the Help System

There are three alternatives for structuring the help system: it may be *panel-by-panel*, *context sensitive*, or *hypermedia-based*.

Panel-by- panel Help: Each panel displayed by the software has a corresponding help panel, invoked when the user makes a request for it. The user request is typically affected by pressing a given function key on the keyboard, or clicking at a help button on the screen.

Context Sensitive Help: Depending on the cursor position at the time of user request, help information, specific to that locality is provided. This approach is difficult to develop and maintain, since there must be close synchronism between cursor location and help information provided. For example, the operating system called System i (formerly OS-400) has a help facility that is completely context sensitive.

Hypermedia-based Help: The help facility is developed as a hypermedia system, invoked by the user making an explicit request (clicking help menu/button or pressing a function key). The user accesses information based on the choices made from a menu, an index or a hyperlink. This approach has become very popular, because it comes close to offering the same conveniences as the context sensitive help, but with much less complexity of design.

Content of the Help System

The content of the help system must be relevant to the software it is designed

for. Quite often, the help system is developed by practicing system documenters, and not the software engineers who design or develop the product. In such circumstances, close coordination is required, if a high quality product is to be the outcome.

Unfortunately, the content of the help system has been a problem area for software systems. Very often, the product documentation that is marketed with the software is voluminous, but inadequate. Different vendors try to document, each producing documents with their own inadequacies. The end result is an inundation of documentation that the user has to scan through in order to get a good handle of the product. The situation is particularly worrying when the product is a software development tool which software engineers need in order to be more proficient on the job.

Ideally, the software engineers should be closely involved in the design and development of the help system (but due to market dynamics, this is often impractical). Alternately, system documenters should have sound appreciation of software engineering.

13.2.2 The User's Guide and System Guide

Traditionally, when a business application is developed for the organization, there used to be a distinction between the user documentation (*user's guide*) and the system documentation (*system manual*). The former was a non-technical document for end users of the system, the latter, a more technical document for the information systems professional. This approach is still relevant for such categories of software. However, as more sophistication is added to the business applications, this distinction is becoming more nebulous.

The user's guide is a non-technical document, suited for the end-users. It contains step-by-step instructions on how to use the software, preferably on a module-by-module basis. The system manual is a technical document that is ideally suited for system managers and software engineers, whose responsibility it will be to maintain the system. It is a technical summary of the system, outlining:

- The main components
- The operating constraints and configuration issues
- Security issues

- Syntax and explanations of system commands (for command interface)
- Explanations of system commands (for menu or graphical interface)

For other categories of software such as development tools for software engineering, this distinction is not relevant. What is required here is a comprehensive set of documents that the user (in this case the software engineer) can use. Typically, the set begins with a *product overview*, then depending on the complexity and scope of the product, there is a document or set of documents for different aspects of the software. This may also be supported by comprehensive (set of) system manual(s). In the era of command based user interfaces, system manuals were very voluminous. Nowadays (in the era of GUIs), they are not as bulky.

Another contemporary trend is to provide the system documentation in electronic form rather than via large volumes as used to be the case. This cuts down on the marketing cost of software engineering companies, allowing for easier packaging, shipping and handling.

Ideally, the software engineers should be closely involved in this aspect of software documentation. Failure to do so often results in poor quality in the documentation of the product. There is no scarcity of software products that have been poorly documented.

Rational Software (now a division of IBM), the company responsible for Rational product line and the Unified Modeling Language (UML), provides a positive example of good software documentation. Most of the product documentation was done by the chief software engineers behind the products — Grady Booch, Ivar Jacobson and James Rumbaugh (see [Rumbaugh, 1999]). The product documentation for Oracle (despite being quite voluminous) also provides an excellent example of good product documentation: it is comprehensive, non-intimidating, easy to use, well organized and packed with good illustrations.

13.3 User Message Management

Undoubtedly, it will be necessary for the operations of the software system to provide user messages to guide the user along. There are three kinds of user messages that a software system may provide:

- **Error Messages** inform the user that an attempted activity is invalid, or an entry is invalid. For example keying in an invalid date, or attempting to access a data item (record) that does not exist, should each elicit a software response that that activity is not permissible at that point in time.
- **Status Messages** inform the user on the current state of an ensuing activity, for example displaying the number of records read from a database file.
- **Warning Messages** alert the user that an attempted activity could result in problems.

There should be some standard as to how and where on the screen, user messages will be displayed. Two possibilities are on the last line of the screen, or in a pop-up window.

In message management, the software engineer specifies how messages will be stored, retrieved and displayed to users of the software. In many scenarios, message management is addressed in predetermined software development standards (review [chapter 9](#)).

13.3.1 Storage and Management of Messages

Two approaches to storing and managing user messages are possible:

- Each application operation stores and manages its own user messages. This is the easy way out. The main problems are:
 - It promotes inconsistencies.
 - It makes the software difficult to maintain.
 - There is no independence between system errors and system applications.
- Store all user messages in a system-wide message file. Messages are given unique identification codes and can be accessed by any application operation. This is the preferred approach and avoids the problems of the first approach.

Obviously, the second approach is preferred to the first, since it provides

more flexibility and control, particularly as the size and complexity of the project increases. It also leads to a more maintainable software product.

13.3.2 Message Retrieval

If the operation-confined approach described above is employed, message retrieval is not an issue. However, if the system-wide approach is employed, then it might be prudent to define and specify an operation to retrieve user messages and return them to calling operations. This retrieval operation would accept an input argument of the message identification code and return it with an additional argument containing the message text.

The calling operation would determine how and where that message is displayed, possibly to established standards ([Chapter 9](#)).

13.4 Design for Real-Time Systems

Not only are computers used to manage information, but also complex manufacturing processes. In many instances, computers are required to interact with hardware devices. The hardware designed in such circumstances is embedded real-time software. Real-time software must react to events generated by hardware and issue control responses that will determine the behavior of the system.

The stages in the design of real-time systems are mentioned below:

1. **Stimulus Identification:** Identify the stimuli that the system must respond to and the appropriate response to each stimulus.
2. **Timing Constraints:** For each stimulus and response, establish a timing constraint.
3. **Aggregation:** Aggregate stimuli into classes (categories). Define a process for each class of stimuli, with allocation for concurrent processes.
4. **Algorithm Design:** Design the required algorithm for each stimulus-response combination. By aggregation,

derive algorithms for processes.

5. **Scheduling:** Design a scheduling system that will synchronize processes according to established (time) constraints.
6. **Integration:** Establish a method of integrating the system into a larger system if necessary, via a *real-time executive*.

13.4.1 Real-Time System Modeling

In modeling real-time systems, the software engineer indicates precisely, all the state transitions (the causes and effects). Techniques used include (review state transition diagrams of [chapter 6](#)):

- Finite state machines (state diagrams)
- State transition diagrams

Since these techniques were introduced earlier ([chapter 6](#)) and will be explored in more advanced courses (OOM as well as Compiler Construction), they will not be discussed further.

13.4.2 Real-Time Programming

Real-time programming remains an exciting (and lucrative) arm of software engineering. Real-time programming is typically done using:

- Assembly language, where the programmer has full control
- Intermediate-level languages such as C, C++, and Ada
- OOPs such as Java and C#

A *real-time executive* is a software component that manages processes and resource allocation in a real-time system. It determines when processes start and stop, and what resources the processes access. The essential components of a real-time execution are:

- A real-time clock

- An interrupt handler
- A scheduler
- A resource manager
- A dispatcher (responsible for starting and stopping processes)

13.5 Design for Reuse

Like in other engineering disciplines, software must be designed with reuse as a given necessity. Hence:

- Where possible, software must be constructed by using tested and proven components.
- New software must be designed so that they can be reused in constructing other software products.

In order for this to be achieved:

- Software engineers must be adequately trained.
- The industry must strive towards a zero-tolerance level for blatant software errors.
- Standardization must take more preeminence than it has taken in the past.
- Software (and components) documentation must be an integral part of software engineering.

Software reuse may be considered at different levels:

- Application systems and subsystems
- System components such as object types (classes) and operations
- Ubiquitous algorithms e.g. sort algorithms, forecasting algorithms, date validation, etc.

- Methodologies

Advantages to reusable software include:

- Improved software quality
- Reduction of development time and cost
- Enforcement of software engineering standards
- Improved reliability of software
- Reduction of risks for new software engineering projects

13.6 System Security

System security has always been, and will continue to be an integral part of software design. There are three levels of security that should be addressed:

- Access to the system
- Access to the system resources
- Access to system data

13.6.1 Access to the System

Access to the system typically involves a login process. Each legitimate user is provided with an account, without which they cannot access the system. [Figure 13-2](#) provides some details that are stored in the user account.

Element	Categorization
Account (login) name	Essential
User name	Essential
User password (usually encrypted and/or disguised)	Essential
User group(s) or class(es)	Optional
User logo or picture	Optional

[Figure 13-2](#). Details Stored About a User Account

Among the categories of software that employ a system level user account

are the following:

- Management Information System
- Strategic information System
- Decision Support System and Executive Information System
- Data Warehouses
- Business Intelligence Systems
- Operating Systems
- Web Information Systems
- Computer Aided Design
- Computer Aided Manufacturing
- Computer Integrated Manufacturing
- Database Management Systems
- CASE Tools and RAD Tools

For these systems, the user accounts are stored in an underlying database file. When the user attempts to log on, this file is accessed to determine whether this is a legitimate user. If the test is successful, the user is admitted in; otherwise, an error message is returned to the user. Of course, the database file must be appropriately designed to store all required details about the user.

Of course, not all software products require the use of system level user accounts as described above. Some products make use of user accounts already defined and stored in the underlying operating system. Others use no system level security at all. Desktop applications and multimedia enabling software are two categories of software that embody this latter approach.

13.6.2 Access to System Resources

Once a user gains access to the system, the next level of security to be addressed is access to system resources. By system resources, we mean database files, source files, commands, programs, services, etc. Depending on how complex the system is, this could be quite extensive, involving the storage and (often transparent) management of a user-resource matrix (also called an *access*

matrix). When the user attempts to access a resource, the access matrix is checked to determine whether the attempted access is legitimate. If it is, the operation is allowed; otherwise, an error message is returned to the user.

One way to implement the resource access matrix is to incorporate its design into the underlying database for the software system, and include related operations for managing it. This approach will become much clearer to you when you study database systems (and learn more about the system catalog) as well as operating systems. Alternately, the matrix may be implemented as an independent system and integrated in the current software system. While the first approach is easier, the latter approach provides more flexibility, and the ability to reuse the resource access matrix in multiple projects.

13.6.3 Access to System Data

If your system involves access of user data in an underlying database, then it might be desirable to manage access to actual data contained in the database. One very effective and widely used methodology for this is the use of logical views. As mentioned earlier, a logical view stores the definition of the virtual database file (table), but stores no physical data. It is simply a logical (conceptual/external) interpretation of data stored in core database files. [Figure 13-3](#) provides an example of a situation requiring logical views. You will learn more about, and develop a better appreciation of logical views after you have completed a course in database systems.

Referring to the CUAIS project, your CUAIS database would most definitely include a **Student** entity that stores basic information about student. Suppose that your college/university has twenty academic departments. You could create a logical view of the **Student** entity for each department. Each view will filter and reorganize data stored in the **Student** entity so that the department chair for that department has read-only access to records of all students in his/her department, but cannot access the information for students in any other department. Rather than giving department chairs unfettered access to the **Student** entity, it provides them with controlled access to information that they need to know, and nothing more.

[Figure 13-3. Example of a Situation Requiring Logical Views](#)

13.7 Summary and Concluding Remarks

Let us summarize what has been discussed in this chapter:

- The system catalog is a data dictionary of the software system. It contains information such as the name, description, and characteristic details of the main system components including information entities and operations. It may also contain definition of system rules.
- With a sophisticated DBMS suite or CASE tool, the system catalog is automatically created and maintained by the software development tool. In a more primitive software development environment, the catalog can be created and maintained as static word processing documents or dynamic files maintained by utility programs.
- Once created, the system catalog should be carefully maintained as it contains useful information about the software system.
- Software documentation typically includes a help facility, a user's guide, and a system guide.
- The help facility may be panel-by-panel, context sensitive, or hypermedia-based. Whichever approach is used, the help facility must be carefully planned.
- The system guide is a technical document, for software engineers and/or managers of the system. The user's guide is a non-technical document for end users. Both should be carefully planned.
- User messages may be error messages, status messages or warning messages. They are designed to assist the user in successfully using the software system.
- User messages may be managed on an operation-by-operation basis, or via a system-wide message file. The latter approach is preferred as it provides more flexibility to the software engineer, and leads to fewer problems during software maintenance.
- Real-time software systems are hardware-intensive systems that operate in real time based on hardware signals and

responses, rather than human intervention. A real-time system passes through the stages of stimulus identification, timing constraints, aggregation, algorithm design, scheduling and integration.

- Real-time system modeling involves extensive use of diagramming techniques such as finite state machines, state transition diagrams, and activity diagrams. Real-time system programming involves low-level programming.
- Code re-use is not magic; it must be carefully planned and managed.
- There are three levels of software system security: access to the system, access to system resources, and access to system data. Thoughtful design of the security mechanism is paramount to the success of the software system.

By following the principles and methodologies in this and the previous four chapters, you are now in a position where you can confidently put together a design specification for your software system. Remember, we are assuming reversibility between phases of the SDLC as well as between stages within any given phase of the SDLC. The alternative to this assumption is to assume the waterfall model; however, as was mentioned in [chapter 1](#), this model is particularly problematic, especially for large, complex systems.

Take some time to review the ingredients of the design specification (see [chapter 9](#)), and the various issues discussed in this division of the text. Then examine the sample design specification of [appendix 10](#) with fresh eyes. You have been armed with the basic skills needed to design quality software! The next three chapters will discuss software development issues.

13.8 Review Questions

1. What details might be stored in a system catalog? How might it be constructed? How might it be used?
2. Briefly discuss three approaches to structuring the help system of a software product.
3. What information is normally provided in the user's guide

of a software product?

4. Describe how you would manage error and status messages for a large software engineering project.
5. Describe the six stages in the design of real-time systems.
6. Describe the three levels of security that many software products are required to address. For each level, outline an effective approach for dealing with security at that level.

13.9 References and/or Recommended Readings

[Foster, 2010] Foster, Elvis C. with Shripad Godbole. *Database Systems: A Pragmatic Approach*. Bloomington, IN: Xlibris Publishing, 2010. See [chapters 13 and 14](#).

[Harris, 1995] Harris, David. *Systems Analysis and Design: A Project Approach*. Fort Worth, TX: Dryden Press, 1995. [Chapter 6](#).

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design 6th ed.* Upper Saddle River, NJ: Prentice Hall, 2005. See [chapters 8 and 15](#).

[Rumbaugh, 1999] Rumbaugh, James, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

[Schneiderman, 2005] Schneiderman, Ben. *Designing the User Interface 4th ed.* Reading, MA: Addison-Wesley, 2005. See [chapter 13](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering 8th ed.* Boston, MA: Addison-Wesley, 2006. See [chapters 15 and 30](#).

PART D



Software Development

This relatively short division of the text involves three chapters:

- [Chapter 14](#) — Software Development Issues
- [Chapter 15](#) — Human Resource Management
- [Chapter 16](#) — Software Economics

The brevity of this division is by no means an indication of the level of importance to be given to software development. To the contrary, software construction remains a critical aspect of the software engineering experience; after all, without development, there is no software product.

Traditionally, software engineering projects would dedicate roughly 20 percent of the time to requirements specification and design specification, and the rest of the time to development. As emphasized throughout the text, this has proven to be an inappropriate and imprudent approach to the discipline of software engineering. Rightly so, much focus has been shifted to the earlier phases of the SDLC, in an attempt to have a more balanced approach, thus leading to the production of higher quality software. This course embraces that shift. Software construction then, ought to be a rewarding, exciting, enjoyable experience of building a product that was carefully, thoughtfully, skillfully and methodically planned. It should not be characterized by guesswork and failure, but confidence and anticipation. After all, you are merely implementing a plan that passed through the rigors of investigation, analysis, design and several iterations of refinement.

It is against this background that you are invited to embark on the software construction experience. Good software construction requires the observance of some management techniques that have either been discussed in previous chapters (for example PERT/CPM), or are too

involved to be thoroughly discussed in this course (for example human resource management and team motivation). It also presumes the mastery of fundamental programming skills. Nonetheless, the chapters pull the critical pieces together to provide you with a comprehensive perspective.

CHAPTER 14



Software Development Issues

This chapter discusses important software development issues that are critical to the successful construction of a top quality software product. It proceeds on the presumption that appropriate steps were taken in the earlier phases of the software engineering experience (namely investigation, analysis and design), and under the following subheadings:

- Introduction
- Standards and Quality Assurance
- Management of Targets and Financial Resources
- Leadership and Motivation
- Planning of the Implementation Strategy
- Summary and Concluding Remarks

14.1 Introduction

If enough thought and planning were put in the design phase, software development should proceed smoothly. In fact, for the meticulous and keen software engineer, software development is fun; it represents the beginning of seeing the fruition of diligent work, invested up front, in the investigation, planning and design of the software. Indeed, the following statement is worth remembering.

Every software engineer knows how to write programs; fewer can excellently design software.

Software development may employ any of the life cycle strategies discussed in [chapter 1](#). In the interest of clarity, these strategies are listed and clarified (in the context of development) below:

- **Waterfall Model:** Keep writing code until the software system is finished.
- **Phased Prototype Model:** Develop the software system chunk by chunk.
- **Iterative Development Model:** Develop the software system iteration by iteration.
- **Rapid Prototype Model:** Take a swipe at developing a working model of the software system and hope for the best; then use user feedback to refine.
- **Formal Transformation Model:** Generate provable code for the software system.
- **Component-based Model:** Develop the software system by integrating used and tested components; write code only for the missing pieces and for integrating the components.
- **Agile Development Model:** Merge the ideas of phased prototyping and iterative development, focusing on the construction and delivery of a product with the minimum required essentials features.

During software development, a team of software developers uses a set of software development tools to write and/or generate code for the software system, based on the design specification. The design specification is used as an input to this phase of the SDLC; the output from this phase is the actual software product (including its documentation).

Management of the project is of paramount importance during this phase. Things may deviate from actual projections and it is here that the management skills of the project manager will be challenged. Accordingly, the project manager will need to exhibit good project management skills; he/she will also need to be creative. Recall that [chapter 8](#) discussed tools such as PERT, CPM, and project management software systems. We will focus on the other related issues in this and the next two chapters.

14.2 Standards and Quality Assurance

In order to ensure that the quality of the software constructed is acceptable, the project manager as well as the software engineers must be concerned about standards and quality. Software quality affects how end users will relate to the product, the level of acceptance the product will receive in the organization, and how the product will perform in the marketplace. As such, the following statement is worth remembering:

Excellent software quality is not magic (though it sometimes seem that way); rather, it's a consequence of sound logic. Excellent software quality will not just happen, if it was not deliberately built into the design, and consistently pursued during development and implementation.

With respect to standards and quality assurance, three issues are worth mentioning:

- The relationship between quality and standards
- The software quality factors
- The quality evaluation

14.2.1 The Relationship between Quality and Standards

During software development, many of the issues addressed in [chapter 9](#) on software development standards are applicable. To review, the following issues will be of importance:

- Naming of database and non-database objects
- Naming of operations
- Programming standards
- User Interface standards
- Documentation standards

- Database design standards

In this regard, the project manager must be knowledgeable and preferably an expert in order to effectively manage the software engineering project. In complex situations he/she may have to lead and manage by example.

Quality assurance (QA) involves the process of ensuring that the software meets the requirements and design specifications, and conforms to established standards. QA is applicable to all aspects of business. [Figure 14-1](#) defines a procedure for maintaining a high software quality during a software engineering project. Essentially, the procedure establishes a loop between software development and assessment against established standards and requirements. Looping continues until the standards and requirements are met.

1. Define and specify requirements
2. Set standards
3. Define development strategies and activities to meet standards and requirements
4. Work to defined standards and requirements
5. Measure the outcome against defined standards and requirements
6. If no problem, go to step 8
7. If problem, identify problems and areas for improvement. Go back to step 4
8. If end, stop; else go to step 4

[Figure 14-1. Procedure for Maintaining Software Quality](#)

In business, a concept that has become very popular since the 1990s, and has remained so, is *total quality management* (TQM). A full discussion of TQM is beyond the scope of this course. Suffice it to say, TQM is a comprehensive philosophy that encompasses all aspects of the organization, and all management functions. Two sound principles of TQM are worth stating here:

- Do it right the first time
- Do it again

The first statement implies that the programmer or software engineer should spend some time to convince himself/herself that a program or module or system works before submitting it for QA evaluation. The QA evaluation then becomes a means of fine-tuning the system rather than identification of major rework.

The second statement implies that the process of proliferation of good standards, models, and methodologies should be pursued. As a practical example,

стандартов, правил и методологий могут не быть ясны. Но в практическом примере, программисту следует потратить некоторое время, чтобы научиться правильно выполнять первую операцию ADD. Это может послужить основой для других операций ADD.

Качество должно быть проактивным, а не реагившим. Иными словами, качество должно стремиться избежать ошибок, а не реагировать на них. Качество должно фокусироваться на всех фазах жизненного цикла системы одинаково, а не на фазе поддержки. Качество должно характеризовать все аспекты программного обеспечения, а не только некоторые из них.

Стандарты качества должны эффективно коммуницироваться на всех уровнях организации. В частности, проектная команда должна быть осведомлена о таких требованиях. Вот одна из риторических вопросов, которые стоит помнить:

What good are standards if no one reads or observes them?

14.2.2 Software Quality Factors

Факторы качества программного обеспечения были впервые введены в [главе 1](#); они также упоминались в [главе 3](#), и снова в [главе 9](#). Их нельзя переоценивать; они влияют на успех или провал программы. Как софтверный инженер, когда вы проектируете программное обеспечение, вы должны постоянно думать о этих факторах. Эти факторы, вместе с вашими стандартами качества, если будут соблюдены во время разработки, помогут достичь высокого качества в итоговой продукции. Для удобства, факторы качества перечислены ниже:

Efficiency, Reliability, Flexibility, Security, User-friendliness, Integrity, Growth Potential, Maintainability, Adequacy of Documentation, Functionality, Cohesiveness, Adaptability, Productivity, Comprehensive Coverage

14.2.3 Quality Assurance Evaluation

Операции, входящие в программное обеспечение, должны быть протестированы отдельно, а также в совокупности. *QA evaluation* (традиционно называемый структурированным обходом) — это комплексный тест программы или компонента программы. Оценка может проводиться на различных уровнях — для операции, группы или связанных операций (модуля), подсистемы или всей программы.

Важные стороны, участвующие в значительной оценке качества, включают следующее:

- Project manager
- Software engineer or programmer responsible for the component
- A principal user
- Someone to take notes (if not the software engineer)

In some circumstances, a tester or a software engineer, prior to a QA evaluation with all the relevant parties, may test the component. The objective of the QA evaluation is to verify that the operation, module, subsystem, or system meets user requirements and conforms to QA standards. Usually, a standard form is developed to record the result of such evaluations. Bear in mind, however, that the forms may be in hard copy, or stored on the computer. [Figure 14-2](#) illustrates what an evaluation form might look like.

<u>Software Quality Assurance Evaluation Report</u>	
Project:	_____
Component:	_____
Operation:	_____
<u>Section A: Conformance to Standards</u>	
<u>Comments:</u>	<u>Action Recommended</u>
1. _____	_____
... _____	_____
n. _____	_____
<u>Section B: User Requirements Assessment</u>	
<u>Comments:</u>	<u>Actions Recommended</u>
1. _____	_____
... _____	_____
n. _____	_____
<u>Section C: Summary</u>	
<input type="checkbox"/> Accept Work	Walkthrough Conducted by: _____
<input type="checkbox"/> Revise Work and Resubmit	Signature: _____
<input type="checkbox"/> Reject Work	Date Filed: ___/___/___

[Figure 14-2.](#) Illustrative QA Evaluation Form

14.3 Management of Targets and Financial Resources

The creativity and management skills of the project manager are challenged most strongly during software development. The critical question to address is, how can resources be allocated in order to ensure that targets are met?

PERT/CPM is project management technique that is commonly used (review chapter 8).

Two additional areas of concern are:

- Budget and expenditure
- The value of the software system

14.3.1 Managing Budget and Expenditure

Budget management is an important aspect of software development. There are two components to budget management: planning the budget and monitoring the expenditure. The budget planning would have been completed from the early (planning) stages of the project. What is required during development is monitoring of the targets, the resources and the expenditure. Following is a summary of what is required in each aspect. However, please note that a comprehensive treatment of budget management is beyond the scope of this course (for additional information, see recommended reading [Morse, 2000]).

Budget Preparation

The budget typically spans a fiscal year. However it might be broken down on quarterly basis or a monthly basis. The budget contains *summary items* and *line items* that make up the summary items. The summary items and some of the line items are typically predetermined, based on the organization's *chart of accounts*.

The budget may also be split into broad categories of expenditure items, for instance *capital expenditures* and *recurrent expenditures*. Capital expenditures relate to investments in fixed assets and/or infrastructure. Recurrent expenditures relate to operational issues (such as salaries, stationary, fuel, heating, electricity,

transportation, etc.)

Figure 14-3 illustrates how a budget might be composed. Please note:

1. Each summary item (e.g. Network Upgrade) would have associated detail line items that may be part of the main document, or included as an appendage. The details will show how each summary amount was arrived at.
2. In some instances, supporting documentation (such as quotations from vendors) may be required.

Division: Information Systems and Services	
Expenditure Areas	Projected Expenditure
Recurrent Expenditure	3,100,000
Salaries	1,400,000
Office Supplies	400,000
Electricity	500,000
Telephone	150,000
Systems Maintenance	400,000
Travel	150,000
Human Resource Development	100,000
Capital Expenditure	400,000
New Information system for Marketing	200,000
Upgrade the Financial Management System	100,000
Network Upgrade	100,000
Total Budget	3,500,000

Figure 14-3. Example of a Budget

Since one's budget will ultimately affect one's ability to pursue the projects of intent, it is imperative that the budget be comprehensive. Also, to avoid embarrassments due to budget overruns, it is always better to over-estimate rather than under-estimate expenditure.

Budget Monitoring

Once a voted budget is in place, the project manager will be aware of this. Software development should proceed according to the budget that is in vogue; ensuring this is the responsibility of the project manager. Issues such as project

crashing (review [chapter 8](#)) or recasting of certain targets may become relevant as the project proceeds. However, note that recasting of targets reflects poorly on the project management, and must only be considered when it is clear that the circumstances warranting these changes are beyond the control of the project manager.

14.3.2 Managing Software Cost and Value

Yet another significant consideration is the actual costing of the software product. This is particularly important if the product is to be marketed, or its value included as part of the organization's capital assets. One popular model for estimating software cost is the *COCOMO model*. This will be discussed in more detail in [chapter 16](#). The basic proposal of the model is that software cost is a function of software size (measured as source code instructions). Of course, other algorithmic cost models have been proposed. Typically, software cost is influenced by the following factors:

- Size (measured in lines of code)
- Complexity (determined by the number of complex calculations and algorithms involved)
- Value added to the organization that acquires the software
- Consumer demand for the product

14.4 Leadership and Motivation

A study of leadership and motivation belongs more appropriately to the field of *organizational theory and behavior* (OTB) and cannot be comprehensively covered in this course. Suffice is to say that the project manager (who might very well be a software engineer) must be appropriately prepared (academically and experientially) to offer effective leadership. He/she must be familiar with various leadership and motivational theories. In passing, a few important points will be stated.

The project manager should know when to apply certain principles and theories of management. Some of the paradigms have been listed in [Figure 14-4](#). These will be clarified in the upcoming chapter.

Autocratic style	Democratic style	Laissez-Faire style
Transformational leadership style	Super leadership style	Task oriented leadership style
Relations oriented leadership style	Goal - Path leadership style	Contingency leadership

Figure 14-4. Management Styles

The ability to adjust to situations and manage accordingly is called situational (contingency) management; this is the preferred approach. The project manager must be able to identify the abilities, needs, and perceived values of members of his/her team and assign activities that will help the respective individuals to realize those needs. This will ensure optimum productivity.

The project manager must be a good motivator. The project manager's most valuable resource is his/her human resource. If you have a team of individuals who are perpetually unmotivated, that team will not achieve much. Software engineering is serious business and must be taken as such.

A full discussion of motivation theory is beyond the scope of this course. **Figure 14-5** provides a brief summary of some prominent motivation theories. Since the software engineer may be often called upon to lead project teams, it is in his/her interest to be cognizant of these theories, in order to be effective at project management.

- **McGregor's Theory X:** This theory purports that workers are only motivated by money. They are basically selfish and lazy and therefore need to be closely controlled and directed. The point to note here is that some people are negative, and are not self-motivated.
- **McGregor's Theory Y:** This theory is the opposite of theory X. It purports that workers are motivated by many different factors apart from money, and that they enjoy their work. The theory argues that given the opportunity, workers will happily take on responsibilities and make decisions for their organization. The point to note here is that some people are self-motivated and welcome challenges.
- **Vroom's Expectation Theory:** Vroom's theory asserts that an employee's motivation to complete a task is a function of the employee's expectation (of the reward), the value placed on such reward, and the instrumentality to accomplish the task. So to get an employee motivated to complete a job, the project manager needs to ensure that the employee is sold on the reward associated with completion of the job.
- **Maslow's Theory:** Maslow's theory of motivation argues that there is a motivational hierarchy that people seek to have satisfied. The hierarchy consists of physical needs, safety needs, love and acceptance, self-esteem, knowledge and understanding, aesthetics, and self-actualization. The needs are fulfilled in that order. The idea here is that the project manager must understand what motivational needs the team member has in order to best know how to motivate him/her.
- **Herzberg's Theory:** This theory submits that worker satisfaction and performance are influenced by two sets of factors. The *hygiene factors* include work conditions, company policy, salary, inter-personal relationships, and salary. These factors do not motivate, but their absence could lead to de-motivation. The *motivation factors* include achievement, growth potential, recognition, status, and responsibility. These factors inspire the employee to perform well. The idea here is that project managers must ensure the presence of all the factors of production in the workplace.

Figure 14-5. Popular Motivation Theories

14.5 Planning of Implementation Strategy

Plans for the implementation must be made and finalized well ahead of the implementation time. This matter will be discussed in [chapter 17](#). The main implementation issues to be considered are

- Operating environment
- Installation
- Code conversion
- Training
- Change over
- Marketing

As you will soon see, these issues could influence the success or failure of the software product in the organization and/or marketplace.

14.6 Summary and Concluding Remarks

Here is a summary of what has been discussed in this chapter:

- If enough thought and planning were invested in the design of the software, actual construction will be an enjoyable, exciting, and rewarding experience. Prudent project management will also be required of the project manager (who is likely to be a lead software engineer) during this period.
- Excellent software quality is not a miracle; it will not just

happen, if it was not deliberately built into the design, and consistently pursued during development and implementation. Procedures for building quality in the software product must be clearly outlined and followed.

- The QA evaluation is used to ensure that established software standards are upheld during the development process.
- During software development, resources, targets and expenditure must be carefully managed to ensure that the project meets its deadlines.
- Leadership and motivation is also a critical factor during software developed. Ideally, the project manager must be an excellent leader and motivator.
- The implementation strategy for the software system must be planned well ahead of the completion of the software development.

The matter of leadership and motivation is extremely important in a software engineering project. For this reason, it is given a bit more attention in the upcoming chapter.

14.7 Review Questions

1. What are the critical issues to be managed during the development phase of a software engineering project?
2. Explain how software standards relate to quality assurance. Outline a procedure for maintaining software quality.
3. Discuss the QA evaluation exercise and propose an instrument for use during this experience.
4. Describe a technique for managing resources, targets, budget and expenditure during software development.
5. How important is leadership during a software engineering project? Explain.

14.8 References and/or Recommended Readings

[Boehm et. al., 1997] Boehm, Barry W., C. Abts, B. Clark, and S. Devnani-Chulani. COCOMO II Model Definition Manual. Los Angeles, CA: University of Southern California, 1997.

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [chapter 16](#).

[Lewis, 1993] Lewis, Phillip V. *Managing Human Relations*. Boston, MA: Kent Publishing, 1993.

[Morse, 2000] Morse, Wayne, James Davis and Al L. Hartgraves. *Management Accounting: A Strategic Approach*. Cincinnati, OH: South-Western College Publishing, 2000. See [chapter 11](#).

[Peters, 2000] Peters, James F. and Witold Pedrycz. *Software Engineering: An Engineering Approach*. New York, NY: John Wiley & Sons, 2000. See [chapters 12 - 14](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [chapters 7 - 9](#).

[Pressman, 2005] Pressman, Roger. *Software Engineering: A Practitioner's Approach* 6th ed. Crawfordsville, IN: McGraw-Hill, 2005. See chapter 26.

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Boston, MA: Addison-Wesley, 2006. See chapters 17-23.

[Van Vliet, 2000] Van Vliet, Hans. *Software Engineering: Principles and Practice*. New York, NY: John Wiley & Sons, 2000. See [chapter 13](#).

CHAPTER 15



Human Resource Management

Human Resource Management (HRM) is arguably, the most important aspect of management in general. This argument can be easily supported, since the most important resource in any organization is the human resource.

In most medium sized and large organizations, there is a human resource director, with responsibilities for the human resource needs of the organization. As indicated in [chapter 1](#), in progressive organizations human resource management is given the same level of priority as information technology.

At every level, managers have as one of their responsibilities, HRM. Software engineers who operate as project managers are by no means excluded. People work on projects, it requires people to administer software systems; people are needed to administer backup procedures, and conduct preventive maintenance, etc. These people will need direction, and in many cases this direction will come from the lead software engineer.

HRM cannot be fully discussed in one chapter (in fact it is treated as a course in many undergraduate and graduate programs). This chapter should therefore not be construed as a substitute for training and education in this area, but rather an overview of the essential aspects of HRM from a software engineering perspective. The chapter will proceed under the following captions:

- Management Responsibilities
- Management Styles
- Developing the Job Description
- Hiring
- Maintaining the Desired Environment
- Preserving Accountability

- Grooming and Succession Planning
- Summary and Concluding Remarks

15.1 Management Responsibilities

The job a manager may be summarized in the following broad activities:

- To motivate people to be the best they can be.
- To seek out the best interest of his or her employees.
- To create the environment that will ensure the achievement of all organizational objectives to which his/her job applies (directly or indirectly).
- To be an example of accountability, professionalism and excellence.

In the pursuit of these objectives, the manager uses various strategies and assumes certain responsibilities (some of which would be indicated on a job description). These strategies and responsibilities will vary according to the organization, the nature of the job, and the individual. Nonetheless, some general functions of management have been identified:

Planning, Organizing, Coordinating, Commanding, Controlling, Hiring

In contemporary literature on management, the trend is to replace the functions of *coordinating* and *commanding* with a single function — *leading*. The rationale is that *leading* is a more friendly term than *commanding* and *coordinating*. Discussion of these basic functions is left to the reader. Suffice it to say that the software engineer should be cognizant of these responsibilities and functions because as mentioned in [chapter 2](#), the job of a software engineer is a management position that often requires supervision of a software engineering team.

15.2 Management Styles

Many theories on management styles have been forwarded by social scientists. A brief survey of the more popular ones follows:

15.2.1 Autocratic Management

In *autocratic management*, the manager dictates to and/or commands his/her team members, and requires them to follow his/her instructions.

Two obvious advantages of this approach are:

- The manager gets things done his/her way.
- The manager has the sense of total control.

Two obvious disadvantages can also be identified:

- The manager's way may not always be the most prudent. The approach presumes that team members are not smart enough to be trusted with major responsibilities. This, we know to be fallacious.
- People do not like to be dictated to; experience has shown that treating people like this appeals to their latent propensity to rebellion, and brings it to the surface. The manager's sense of control may therefore be false.

15.2.2 Egalitarian (Democratic) Management

In *democratic management* the managers solicits ideas and inputs from his/her team members. The best suggestions are taken and incorporated into planning and operation exercises.

Advantages of this approach are as follows:

- Team members are encouraged to participate in important decision-making. This gives them a sense of belonging and commitment.
- The sense of belonging and commitment motivates team members to share in the corporate mission and consider going the extra mile in pursuit of it.

- Healthy work relationships can be built between managers and team members.

The main disadvantage of this approach is that does not always yield positive results; experience has shown that there are scenarios that do not warrant it.

15.2.3 Laissez Faire Management

In the *laissez faire* approach, the manager assumes little or no control. He/she allows team members to do whatever they please.

Although there are situations that warrant this management style (for example: a Christmas party), in most cases it results in chaos, and lack of achievement of significant objectives.

15.2.4 Path-Goal Leadership

In *path-goal leadership*, the leader influences performance, satisfaction and motivation of his/her team members by

- setting clear achievable goals for team members;
- offering rewards for achieving these goals;
- clarifying paths towards these goals;
- removing obstacles to performance and eventual achievement of goals.

The leader does this by adopting a certain leadership style based on the situation:

- **Directive Leadership:** This can be done by specifying and assigning objectives (goals), strategies and tasks in pursuit of the established objectives, providing advice and direction to team members.
- **Supportive Leadership:** This is done by building good relationships with team members.
- **Participative Leadership:** Decision-making is based on

consultation with team members.

- **Achievement-oriented Leadership:** The leader sets challenging goals and high performance expectations. Much confidence is expressed in the group's ability. This inspires the members to give their best.

Advantages of the approach are

- It builds confidence of team members and inspires them to contribute their best.
- It can build a healthy working environment.

Disadvantages are

- The environment could become overly competitive, thus creating animosity among team members.
- If team members are not adequately prepared for this, the efforts could be counterproductive.

15.2.5 Transformational Leadership

In transformational leadership, the leader presents himself/herself as an agent of change (presumably for the better). A strong relationship is built between leader and followers. Transformational leadership often involves a vision to forge an organization in a new direction and elicit change. It is characterized by strong ideas, inspiration, innovation, and individual concerns.

The main advantage of this leadership style is that followers are “fired up” to effect the required changes as enunciated by the leader. There is a strong sense of commitment to set goals.

The disadvantages are

- If change does not come in a timely manner frustration could overtake some of the team members.
- The approach begs the question, what happens after the desired changes are achieved?

15.2.6 The Super Leader Approach

In the *super leader* approach, the leader sets himself/herself up as an icon to be emulated by team members. By example, he/she sets high standards, and challenges team members to emulate them. The approach also addresses the matter of succession planning (to be discussed later) by grooming selected team members to be super leaders at various levels.

The advantages of this approach are the following:

- The approach promotes the idea of leadership by example, a principle that resonates well with people, and traces back to biblical history (characters such as Jesus Christ, Moses, and Joshua are described in Judeo-Christian literature as super leaders), and forward to current times (you can do doubt identify super leaders in your workplace, social/professional affiliation, or country).
- Like path-goal leadership, the super leader approach motivates people to be the best they can be.

The main challenge of this approach is that the super leader must be well prepared and versed in the activities that he/she desires the team members to engage in.

15.2.7 Task-Oriented Leadership

In *task-oriented leadership*, the leader's primary focus is the conducting of activities, in pursuit of established goals. These activities must be done at whatever cost.

The main advantage of the approach is that it is achievement-oriented, and is therefore likely to produce a high level of productivity.

The main disadvantages of the approach are as follows:

- The leader could become insensitive to the human needs of team members, while being absorbed with pursuing his/her objectives.
- This problem could result in a demoralized, de-motivated team, thus inhibiting the leader's ability to achieve the very

goals being pursued.

15.2.8 Relation-Oriented Leadership

A *relation-oriented leader* seeks, as a primary focus, to build good relationships with team members. The thinking behind this is that if team members have a good relationship with their leader it will inspire them to perform.

The main advantage of this approach is that when it works, the results are very convincing. One possible reason for this is that team members feel a strong sense of belonging, and ownership of the project.

The main disadvantages of the approach are as follows:

- The approach does not always work. In fact it is possible for team members to enjoy good relationships with their leader, and still not perform well.
- The worst-case scenario of this approach is that the team becomes a social club where there is much fellowship, but little work.

15.2.9 Contingency Leadership

Contingency leadership theory is an argument for pragmatism: Since the individual approaches all have their advantages and disadvantages, the manager should reserve the right to employ different approaches, depending on the scenarios that present themselves.

The main advantage of contingency leadership is that the weaknesses of any approach are avoided while capitalizing on the strengths of the respective approaches.

The main disadvantage of the approach is that inexperienced managers could make bad judgments about which strategy to employ. However, with experience, contingency managers usually make excellent decisions on the average.

15.3 Developing Job Descriptions

As a computer science professional with management responsibilities, you may

be called upon develop job descriptions for junior positions in your division or department, from time to time. Different organizations will have different standards regarding how their job description should be written.

[Figure 15-1](#) provides a checklist of the essential components of a job description. For an example, please review [Figure 2-1 of Chapter 2](#).

- **Company Heading:** Name of the company
- **Job Summary:** Overview of the job
- **Main Functions:** Itemized list of salient responsibilities
- **Related Authority:** What can be done independent of prior consultation
- **Reporting Relationship:** Who does the person report to?
- **Required Qualifications:** Minimum and preferred qualifications
- **Other Special Requirements:** Other expectations not previously mentioned

[Figure 15-1. Basic Components of a Job Description](#)

■ **Caution** In many cases you will find that job descriptions you are looking for are either in need of improvement, or are nonexistent. In either case, your task must be to leave the situation in a better state than you found it.

15.4 Hiring

Hiring is a very important management function. No manager can function without people (human resource). The software engineer may be called upon to participate in the process of hiring suitably qualified individuals to be part of the project team.

Below are some important considerations for the hiring process:

1. Clearly define the position to be filled. If it is a new one, then approval from a senior level of management may be required. In any case a clearly defined job description should be in place.
2. Advertise for applicants to fill the position.
3. Convene an interview panel.

4. Arrange for interviews of the applicants. Depending on the organization, the interview schedule may vary. Typically, job interviews are done in three stages: Firstly, a technical interview scrutinizes the technical and professional preparedness of the applicant. Next, a human resource interview looks at the overall individual and tries to determine whether they would be suitable for the job. A final interview is usually done to make an offer to the selected person. Please note that in some instances, the stages may be merged. For instance, technical and human resource interview may be merged as one.

- A technical interview scrutinizes the technical and professional preparedness of the applicant.
- A human resource interview looks at the overall individual and tries to determine whether they would be suitable for the job.
- A final interview is usually done to make an offer to the selected person.
- In some instances, the stages may be merged. For instance, technical and human resource interview may be merged as one.

5. Select the most suitable individual.

Before conducting an interview, the software engineer must make the required preparation:

1. Prepare a set of criteria to be met by the incumbent.
2. Review the applicant's curriculum vitae.
3. Prepare questions that are consistent with the defined criteria.

It is standard practice to have in each organization, an interview evaluation form. This may vary with different departments, as well as with the positions being considered. Obviously, for SE/IT jobs, the interview evaluation forms must be prepared by the SE/IT executive in charge, and given to members of the interview panel. After each interview, the panel members complete their

respective evaluations of the applicant. [Figure 15-2](#) illustrates a sample interview evaluation form.

LMX Software Inc. Interview Evaluation Form, Web Technology Department		
Interviewee:	<hr/>	
Position Considered:	<hr/>	
Date of Interview:	____/____/____	
Interviewer:	<hr/>	
<u>Criteria</u>	<u>Comment</u>	<u>Evaluation Score</u>
Required Qualification:	<hr/>	[] of []
Professional Preparation:	<hr/>	[] of []
Familiarity with Required Technology:	<hr/>	[] of []
Human Relations Skills:	<hr/>	[] of []
Problem-Solving Skills:	<hr/>	[] of []
Motivation and Drive:	<hr/>	[] of []
Recommendation:	Hire <input type="checkbox"/> Do not Hire <input type="checkbox"/>	
Comment:	<hr/>	

[Figure 15-2](#). Sample Interview Evaluation Form

In conducting the interview, below is a checklist of some of the things you should probe for each candidate:

- **Required Qualification:** This may require asking questions about the institutions attended by the candidate, and the courses pursued.
- **Professional Preparation:** This may require asking questions relating to prior working experiences of the candidate.

- **Familiarity with Technology:** This may require asking technical questions relating to the technologies and methodologies relating to the job.
- **Human Relations Skills:** This may require asking questions relating to the candidate's handling of human relations challenges in the past, as well as simply observing how the candidate handles pressure.
- **Motivation and Drive:** Here you try to assess the candidate's enthusiasm for the job.
- **Problem-Solving Skills:** Is the candidate adept at solving various problems on short notice? Here, problem scenarios are posed to the interviewee, and he/she is asked to describe how they would address the various problems.

15.5 Maintaining the Desired Environment

Having a team in place is good. But how do you create and maintain the working environment that you desire? And how do you get the team to cooperate and support you? This is one of the challenges of management. There are no straight answers to these questions, but there are guidelines. If, as the team leader, you are coming into this position due to a promotion, your challenge may be different from the situation where you are coming in as a new addition to a team. The spectrum of possible human reactions ranges from jealousy and vindictiveness on one extreme, to complete commendation and adulation on the other.

Whether you got a promotion, or were hired into the position, you are likely to face initial challenges. Following are some uncomfortable situations that you may face:

- There may be animosity in the camp as to whether you were most deserving of the position. This possibility is increased if members of the team were considered for the position and then bypassed.

- There may be some resistance to changes that you want to put in place, in your new capacity.
- Individuals may try to challenge your mettle during the early period of your administration.

As the leading software engineer, you want to create an environment where negative factors that could potentially undermine the success of your team are discouraged, and positive factors are encouraged and reinforced. With this focus in mind, it is a good idea to schedule an initiation meeting, shortly after assuming your responsibilities. At this meeting the following activities should occur:

1. Meet the team members (if the team is very large, then meet the key players, for instance people who report to you, along with the supervisors).
2. Clearly outline what your expectations are, and perhaps the mode of operation that you will pursue.
3. Find out what the team members expect from you, and determine whether you can deliver on those expectations.
4. If you are new to the organization, try to get an appreciation of each (major) team member's job (prior to the meeting, you should familiarize yourself with the company norms and policies).

In order to achieve and maintain the desired environment, the following strategies will be also useful:

- Be a good motivator.
- Develop good conflict resolution skills.
- Be an effective communicator.
- Be generous on rewarding outstanding achievement, and consistent in treating errant actions.

Motivation was discussed in the previous chapter; we will address the other issues here.

15.5.1 Effective Communication

Successful managers are usually effective communicators, both orally and in written form. This is also true for software engineers. Having prepared the requirements specification and design specification for your software engineering project, it will be imperative that team members are sold on the project. If the team members were involved during the preparatory work, then this should not be difficult.

Below are a few experiential tips on effective communication:

- Have the mission statement of the organization and that of the division or department attractively framed and strategically positioned in each office.
- Consider putting beautifiers (e.g. potted plants) in various offices. Plants are known to provide therapeutic value and natural beauty to corporate offices.
- Consider strategically placing memory gems to remind employees of their responsibilities. Below are two examples:

Excellence Begins Here...

Excellence is our Standard Requirement

- Avoid being perceived as antagonistic or confrontational.
- Involve team members in the determination of development strategies so that they can experience ownership of the plans.

15.5.2 Conflict Resolution

If you are successful at being an effective communicator, this will pave the way for minimizing conflicts and resolving them. Conflict, by definition, is not necessarily a bad thing, in fact, it can serve to provide different perspectives to a problem, and this leads to a more comprehensive solution. What you want to discourage is employees becoming too personal over their disagreements.

The following approaches ([Figure 15-3](#)) may be employed in resolution of

conflicts. Although they offer no guarantees, you should often find them useful.

Strategy	When to Use
Ignore	Conflict is between two members and it does not threaten the team's success
Confront	Conflict threatens the team's success
Assign someone else to resolve	Conflict is between two members and does not threaten the team's success.

Figure 15-3. Conflict Resolution Strategies

In confronting a conflict, consideration should be given to the source of the conflict:

- If the conflict is about personal feelings of team members and yourself, be prepared to apply some empathy, or to be conciliatory.
- If the conflict is about work (e.g. how to approach a problem) be objective and impartial.
- If the solution sought does not seriously threaten the success of the project, be prepared to make or accept compromises.

15.5.3 Treating Outstanding Achievements and Errant Actions

It is a healthy practice to have in place a system that recognizes outstanding achievement, and discourages errant actions. This must be communicated to all the team members so that there are no surprises. Better yet, if the employees participated in determining the rules and consequences, they will not feel cheated of victimized when they are applied.

Reward for outstanding achievement may come in different forms, for example adding the employees name to a prestigious list of achievers, providing a special gift, etc. Discipline for errant actions is usually in the form of a letter of reprimand, suspension, or termination of employment.

15.6 Preserving Accountability

It is the responsibility of every manager to put in place a system of

accountability which ensures that team members perform according to expectations. This will ultimately ensure that important deadlines are met and goals are accomplished. The software engineer is by no means excluded from this responsibility.

In preserving accountability, the software engineer may pursue a number of strategies:

- Design and assign the work schedules of team members.
- Have a system of evaluating performance against assigned work.
- Have a system of reward/recognition for outstanding performance (as discussed in the previous section).

15.6.1 Designing and Assigning Work

As software engineer in charge of a project, you must be able to define work, provide clarification and motivate team members to perform the desired activities. *Management by objectives* (MBO) is a term often used to describe the situation where employees are set clear objectives over an evaluation period, and are then evaluated on those objectives.

The work assigned must align with the corporate objectives as well as established strategies of your division or department. The following guidelines should be useful:

- Each employee must be assigned work that is commensurate with his/her job, level of expertise and professional capability.
- Deadlines must be clearly established.
- Deadlines must be realistic.
- The employee must agree to both the assigned work and the deadline, in a cordial discussion. If there are differences, they must be resolved.
- The employee must be made to clearly appreciate how their work fits into the big picture of the department, division and by extension, the organization.

- If required, there must be clearly established checkpoints to different activities on the employee's work schedule.

[Figure 15-4](#) illustrates a sample work schedule of a team member on a software engineering project.

Note that the assignment clearly identifies the project, the specific activities, and target date(s).

LMX Software Software Development & Research Department Software Engineering Assignment
Name: Jacob Lambert
Project: CUAIS
Activities:
1. Gather information for Finance, Payroll & Insurance. 2. Prepare Requirements Specification for areas above. 3. Prepare Requirements Specification Documents for all areas 4. Prepare Design Proposal for Finance.
Due Date: February 27, 2006
Status on Due Date: _____
Decision Taken: _____
Supervisor: _____

[Figure 15-4.](#) Sample Work Schedule

15.6.2 Evaluating Performance

Performance evaluation is typically done at the end of the determined evaluation period. Typically, companies have monthly, quarterly, and annual evaluations.

If the organization is on an MBO program, then the annual evaluation is usually significant, since it might mean a large or small bonus, depending on how favorable of the evaluation is. Here are a few important points about performance evaluation:

- The evaluation must be based on the established objectives and assigned activities that were agreed upon with the

employee at the start of the evaluation period.

- The evaluation must be objective, and with supportive evidence.
- The evaluation must be signed by both appraised (i.e. the employee) and appraiser.
- Where the employee might have missed established targets, or performed below expectations, measures must be put in place to help the individual to improve on the next appraisal.

[Figure 15-5](#) illustrates an employee appraisal. Note that the appraisal clearly identifies the evaluation criteria, the assessment period, comments and signature by the appraiser as well as the appraised.

LMX Software
Software Development & Research Department
Employee Evaluation

Name & Title:	Jacob Lambert, Software Engineer	
Supervisor & Title:	Elvis Foster, Software Development Manager	
Appraisal Period:	January – March 2006	
Appraisal Criteria	Comments	Ratings
Availability		
Punctuality	Seldom late	4/5
Availability for Emergencies	Usually available	5/5
Attendance	Needs improvement	4/5
Performance on the Job		
Ability to work on own initiative	Manages on his own	5/5
Productivity level	Needs to improve throughput	4/5
Capacity to Follow Instructions	Excellent	5/5
Capacity to see Jobs to Completion	Excellent	5/5
Completion of Task on Time	Needs improvement	4/5
Dependability	Very dependable	5/5
Professional Development		
Attitude to New Concepts/Challenges	Excellent	5/5
Capacity for Critical Thinking	Excellent	5/5
Willingness to Learn	Excellent	5/5
Attitude to Authority	Excellent	5/5
Human Relations		
Relationship with Peers	Gets along with everyone	5/5
Relationship with Team Members	Excellent	5/5
Relationship with Superiors	Excellent	5/5
Communication Skills	Could be improved	4/5
Dress and Deportment	Excellent	5/5
Overall Evaluation		85/90 = 94.4%

Authentication

Appraiser's Comment: Excellent employee who needs to improve in areas identified.

Appraisee's Comments: I think the evaluation is accurate. I will work on the weaknesses identified.

Employee's Signature: _____ Date: _____

Supervisor's Signature: _____ Date: _____

Figure 15-5. Sample Employee Appraisal

15.7 Grooming and Succession Planning

Succession planning is a very important function of any conscientious manager; it is particularly critical in the case of information management, since any failure of the organization's information systems could prove disastrous. A responsible software engineer will therefore have a small cadre of (one to three) individuals (depending on the size of the division or department) who are specially trained and prepared to assume overall responsibility, should he/she temporarily or permanently depart the organization. Although selfishness often inhibits this being done, prudence and professionalism demand it.

A few points to note:

- The super leader and path-goal leadership styles both facilitate the principles of succession planning, without any additional effort on the part of the manager.
- The individuals being groomed need not know that they are in fact being specially prepared for possible takeover; this knowledge sometimes creates tensions between (or among) the contenders.
- Implementing a succession plan is not usually easy. In many instances, there are fallouts that might result in some individual(s) leaving the organization. Case in point: When General Electric's CEO and chairman stepped down (in 2000) and named his successor, the other two contenders immediately started to make arrangements to leave the company.

Despite possible negative repercussions from implementing a succession plan, it remains a good idea as the benefits far outweigh the possible drawbacks. Among the benefits to be gained are the following:

- Continuity on mission critical projects, even in the face of sudden unavailability of the project manager
- High level of motivation among team members

15.8 Summary and Concluding Remarks

It is now time to summarize what has been covered in this chapter:

- The functions of a manager may be summarized as planning, organizing, coordinating, leading and hiring. Software engineers with management responsibilities are often called upon to carry out these functions.
- Among the leadership styles available to the software engineering project manager are the following:
 - autocratic style; democratic style; laissez-faire style; goal-path leadership style; transformational leadership style; super leadership style; task oriented leadership style; relations oriented leadership style; contingency leadership style.
- The software engineer should be comfortable updating or creating job descriptions for himself/herself or other members of the software engineering team.
- The software engineer may be called upon to engage in the process of hiring new members of the software engineering team. Knowing the standard approach for this function is therefore important.
- The software engineer should know how to create and maintain an environment that is conducive to the success of the software engineering projects. This includes effective communication, excellent conflict resolution, recognizing outstanding achievements and discouraging errant behavior.
- The software engineer should know how to preserve accountability in the workplace. This includes designing and assigning work, as well as evaluating performance of team members.
- A responsible and smart software engineer always has a grooming and succession plan.

Another issue that needs to be addressed during development and refined over the effective life of the software system is the matter of software economics. This will be discussed in the next chapter.

15.9 Review Questions

1. What are the main responsibilities and functions of a manager? Why are they relevant to software engineering?
2. What are the different management styles that are available to the software engineer? For each style, identify the advantages and disadvantages; also identify a scenario that would warrant the application of this approach?
3. Assume that you are a software engineer with the responsibility of managing a software engineering project. Develop a job description for a programmer on your team.
4. What criteria should guide your assessment of a prospective team member?
5. If you were a software engineer in charge of a software engineering project, what principles would guide your effort to create and maintain a productive and congenial working environment?
6. If you were a software engineer in charge of a software engineering project, what principles would guide your effort to create and maintain accountability in the workplace?
7. What are the pros and cons of implementing a succession plan?

15.10 References and/or Recommended Readings

[Lewis, 1993] Lewis, Phillip V. *Managing Human Relations*. Boston, MA: Kent

Publishing, 1993.

[Sayless, 1989] Sayles, Leonard R. *Leadership: Managing in Real Organizations* 2nd ed. New York, NY: McGraw-Hill, 1989.

[Reece, 1987] Reece, Barry L. and Rhonda Brandt. *Effective Human Relations in Organizations* 3rd ed. Boston, MA: Houghton Mifflin Company, 1987.

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Boston, MA: Addison-Wesley, 2006. See Chapter 25.

CHAPTER 16



Software Economics

Software economics was first introduced in [chapter 3](#) (section 3.7) though not by that term. At that time, we were discussing the feasibility of the software engineering project. After reading [chapter 3](#), one may get the impression that software cost is equal to development cost. In this chapter, you will see that the two are often different; that development cost is just one component of software cost; and that there are other factors. You will also see that determination of software cost, software price and software value are difficult issues that continue to be the subject of research and further refinement. The chapter proceeds under the following captions:

- Software Cost versus Software Price
- Software Value
- Assessing Software Productivity
- Estimation Techniques for Engineering Cost
- Summary and Concluding Remarks

16.1 Software Cost versus Software Price

There are few cases where software cost is the same as software price; in most cases, they are different. Let us examine both.

16.1.1 Software Cost

As mentioned in [chapter 3](#) (section 3.7.4), there are many components that go into the cost of developing a software system. These cost components are summarized in [Figure 16-1](#).

Cost Component	Comments
Equipment Cost	Made up of hardware cost, software cost, cost of transporting equipments, depreciation over the acquisition and usage period.
Facilities Cost	Includes computer installation cost, cost of electricity & cooling, security cost
Engineering Cost	Includes investigation, analysis & design cost; development cost; implementation cost; training cost
Operational Cost	Consists of cost of staffing; cost for data capture and preparation; supplies and stationery cost; maintenance cost

Figure 16-1. Software Engineering Cost Components

Software engineering is a business as well as an applied science. The business approach that many organizations employ is simply to put a price tag on each of these components based on experience as well as business policies: The equipment cost, facilities cost and operational cost will continue to remain purely business issues. The standard business approach for determining the engineering cost is to multiply the organization's prescribed hourly rate by the estimated number of hours required for the project. However, as you will see later, there have been efforts to apply more scientific and objective techniques to the evaluating engineering cost through the exploration of deterministic models.

16.1.2 Software Price

Determining the software cost is important but it does not complete the software evaluation process. We must also determine the software (selling) price. Naturally, the biggest contributor to software price is software cost. Three scenarios are worth noting:

- If your organization is a software engineering firm and/or the product is to be marketed, then determining a selling price that consumers will pay for the product is critical. In this case, the price may be significantly discounted since software cost can be recovered from multiple sales.
- If your organization is not a software engineering firm

and/or the product will not be marketed to the consuming public, then the software price is likely to be high, since its cost cannot be recovered from sale. In this case, the minimum price is the software cost.

- If your organization is a software engineering firm contracted to build a product exclusively for another organization, then the software price is likely to be high, since the software engineering firm must recover its costs and make a profit.

[Figure 16-2](#) summarizes some of the main factors affecting software pricing. Please note that there may be other factors that affect the pricing of the software (for instance other quality factors). The intent here is to emphasize that pricing a software system is not a straightforward or trivial matter.

Market Opportunity: The selling price may be significantly discounted in order to break into a new market. The software engineering firm may anticipate huge benefits in other areas of the market, and therefore make huge pricing concessions on the product. You will recall the recent rivalry between Microsoft and Netscape, and how it ended: Microsoft bundled its Internet Explorer software for free with its Windows operating system. Netscape protested. After a contemptuous court battle, Netscape won the battle but Microsoft won the war. Today, Netscape is no longer around, while Microsoft remains the world's largest software engineering firm.

Uniqueness: The more unique the product, the higher the price is likely to be.

Usefulness and Quality: The larger the anticipated use of the product, the lower the price is likely to be. However, this may be offset by packing more features into the product. The price of the software is also a function of its quality (and whether this quality is known by potential users). A good case in point is IBM products. In many cases, the consuming public is asked to pay more for a product from IBM than they would for the same product from another company. This has not significantly affected IBM sales, because generally speaking (and irrespective of your like or dislike of the company), when you purchase an IBM product, you are getting a product of high quality.

Contracting Terms: If the software engineering firm was contracted to build the product for a particular customer, then if the customer retains ownership of the source code, its price significantly increases. On the other hand, if the software engineering firm can retain the source code and use it for other purposes, the price is significantly reduced.

Requirements Volatility: If the requirements are likely to change over the short or medium term, a software engineering firm may lower its bidding price in order to win a contract, and include a clause that treats changes in the requirements as a separate price item.

Financial Health: Generally speaking, software engineering firms that are financially strong are more reluctant to lower their prices than those that are not. Of course, there are exceptions to this observation. In recent years, the consuming public has benefited from huge concessionary overtures from leading software engineering firms such as Oracle, Microsoft, and IBM.

Software Engineering Cost: As clarified in figure 16.1.

Figure 16-2. Some Factors Affecting Software Pricing

16.2 Software Value

The value of a software system may be different from the price. Whether your organization intends to market the software or not, it is often useful to place a value on the product, for the following reasons:

- If the product is being marketed, placing a value on it can provide a competitive advantage to the host organization (i.e. the organization that owns the product) if the value is significantly more than the price. The host organization can then use this as marketing advantage to appear generous to its prospective consumers.
- If the product is being kept by the host organization, then having a value that is significantly higher than its price/cost makes the acquisition more justifiable.
- Whatever the situation, it makes sense to place a value on the product from an accounting point of view.

The big question is, how do we place a value on a software product? There is no set formula or method for answering this question; therefore we resort to estimation techniques. A common sense approach is to assume that software value is a function of any or each of the following:

- Software cost
- Productivity brought to the organization (and how this translates to increased profit)
- Cost savings brought to the organization

In the end, determining a value for the software system is a management function that is informed by software engineering. For this reason, we will examine some of the estimation techniques that are available (section 16.4). Before doing so, we will take a closer look at evaluating software productivity.

16.3 Evaluating Software Productivity

There are two approaches to assessing software productivity. One is to concentrate on the effort of the software engineer(s). The other is to concentrate on the added value to the enterprise due to the software product. Most of the proposed models tend to concentrate on the former approach. In keeping with the theme of the text, and in the interest of comprehensive coverage, we will examine both approaches.

A common method of assessment of software productivity is to treat it as a function of the productivity contribution of the software product, and the collective engineering effort. Two common types of metrics have been discussed in [Sommerville, 2006]. They are *size-related metrics* and *function-related metrics*. A third approach is a *value-added metrics*; it attempts to determine and associated value added by a software system to an organization. A brief discussion of each approach follows.

16.3.1 Size-related Metrics

Size-related methodologies rely on the software size as the main determinant of productivity evaluation. Common units of measure are the number of lines of source code, the number of object code instructions, and the number of pages of system documentation. To illustrate, a size-related metric may compute a software productivity index for a project, based on the formulae in [Figure 16-3](#).

$$PI\{Code\} = (LOC)/(DT)$$

Key:

1. $PI\{Code\}$ is the productivity index based on code. It is expressed in lines of code per programmer-month i.e. LOC/pm.
2. LOC is the number of lines of code.
3. DT is the development time, expressed in programmer-months (pm). A programmer-month is the period of one month (assuming 8-hour day) for one programmer.

$$PI\{Doc\} = (POD)/(DT)$$

Key:

1. $PI\{Doc\}$ is the productivity index based on documentation. It is expressed in pages of documentation per programmer-month i.e. POD/pm.
2. POD is the number of pages of documentation.
3. DT is the documentation time, expressed in programmer-months (pm). A programmer-month is the period of one month (assuming 8-hour day) for one programmer.

Figure 16-3. Formulae for Size-related Metrics

Associated with this approach are the following caveats:

- If different programming languages are used on the same project, then making a single calculation for $PI\{Code\}$ would be incorrect, since the level of productivity varies from one software development environment to another. A more prudent approach would be to calculate the $PI\{Code\}$ for each language and take a weighted average.
- Most of the documented size-related metrics concentrate on lines of code, with scant or no regard for documentation. This plays squarely into the fallacy that software engineering is equivalent to programming. The lower half of [figure 16-3](#) has been added to provide balance and proper perspective to the analysis. If as proposed by this course, more effort ought to be placed on software design than on software development, and that the latter should be an exciting and enjoyable follow-up of the former, then any evaluation of software cost or productivity should reflect that perspective.

The main problem with this model is that it does not address the important matter of software quality. What if the software product and documentation are both voluminous due to faulty design? The model does not address this concern.

16.3.2 Function-related Metrics

Function-related methodologies rely on the software functionality as the main determinant of productivity evaluation. The common unit of measure for these methodologies is the *function-points* (FP) per programmer-month (a programmer-month is the time taken by one programmer for one month, assuming a normal work week of 40 hours). The number of function points for each program is estimated based on the following:

- External inputs and outputs
- User interactions
- External inquiries
- Files used

Additionally, a complexity weighting factor (originally in the range of 3 to 15) is applied to each program. Next, an *unadjusted function-point count* (UFC) is calculated by cumulating the initial function-points count times the weight, for each component:

$$\text{UFC} = \sum [(\text{FC}) * (\text{W})] \text{ where FC is the function-point count for a program component and W is the complexity weight for that program component.}$$

Next, complexity factors are assigned for the project based on other factors such as code reuse, distributed processing, etc. The UFC is then multiplied to this/these other complexity factor(s) to yield an *adjusted function-point count* (AFC). Finally, the productivity index is calculated.

[Figure 16-4](#) summarizes the essence of the approach. Associated with this approach are the following caveats:

- The approach is language independent.
- It is arguable as to how effective this approach is for event-driven systems and systems developed in the OO paradigm. For this reason, *object-points* have been proposed to replace function points for OO systems. We will discuss object-points later (section 16.4.3).
- The approach is heavily biased towards software

development rather than the entire software engineering life cycle.

- The approach is highly subjective. The function-points, weights, and complexity factors are all subjectively assigned by the estimator.

1. Determine the FC and W for each component
2. $UFC = \sum[(FC) * (W)]$
3. Identify other complexity factors for the project ... C1, C2, ...Cn
4. $AFC = UFC * C1 * C2 * \dots * Cn$
5. $PI = (AFC)/(DT)$

Key:

1. FC is the function-point count for a program component and W is the weight for that component.
2. UFC is the unadjusted function-point count.
3. AFC is the adjusted function-point count.
4. PI is the productivity index expressed in function-point counts per programmer-month i.e. FC/pm.
5. DT is the development time, expressed in programmer-months (pm).

Figure 16-4. Calculations for Function-related Metrics

The matter of software quality still remains a concern, though to some extent, it has been addressed in the software's functionality. Indeed, it can be argued that to some extent, a software system's functionality is determined by the quality of the software design.

16.3.3 Assessment Based on Value Added

There is much work in the area of value-added assessment in the field of education as well as other more traditional engineering fields. Unfortunately, the software engineering industry is somewhat lacking in this area. In value-added software assessment, we ask and attempt to obtain the answer to the question, what value has been added to the organization by introduction of a software system or a set of software systems? In pursuing an accurate answer to this question, there are two alternatives that are available to the software engineer:

- Evaluate the additional revenue that the software system facilitated.
- Evaluate the reduced expenditure that the software system has contributed to.

These alternatives are by no means mutually exclusive; in fact, in many instances they both apply. One way to conduct the analysis is to estimate the useful economic life of the software system, and compute the above-mentioned values over that period. In hindsight, this may be challenging, but by no means insurmountable. However, the reality is, in most cases, it is desirable to conduct this analysis prior to the end of the economic life of the system. Moreover, in many cases, this analysis is required up front, as part of the feasibility study for a software engineering venture (review section 3.7).

[Figure 16-5](#) provides two formulae that may be employed in estimating the value added by a software system. The first facilitates a crude estimate, assuming that interest rates remain constant over the period of analysis (of course, this is not practical, which is why it's described as a crude estimate). Since this approach involves taking the difference between the value added and the acquisition cost, you may call it the *difference method*. The second formula computes the *net present value* (NPV), with due consideration to interest rates; it is considered a more realistic estimate. The simple adjustment to be made here is to ensure that the *cash flow per annum* includes additional revenue due to the system as well as reduced expenditure due to the system.

$$SV = [(EVA) - (AC)]/(EL)$$

Key:

1. SV is the software value in dollars.
2. EVA is the estimated value added (which is increased revenue + reduction in expenditure).
3. AC is the acquisition cost of the software.
4. EL is the estimated economic life of the software in years.

$$NPV = -A_0 + A_1 / (1+r_1) + A_2 / (1+r_2)^2 + \dots + A_n / (1+r_n)^n$$

Key:

1. NPV is the net present value. It allows you to pull forward into the present, the future value of the software system after n years.
2. A_0 is the initial investment.
3. A_i is the annual cash flow (which is increased revenue + reduction in expenditure).
4. r_i is the rate of interest.

[Figure 16-5. Estimating Software Value-added](#)

16.4 Estimation Techniques for Engineering Cost

In the foregoing discussion, the importance of the engineering cost and the challenges to accurately determining it were emphasized. In section 16.1, it was mentioned that the standard business approach to estimating engineering cost is to multiply the estimated project duration (in hours) by the organization's prescribed hourly engineering rate. In this section, we will examine the engineering cost a bit closer, and look at other models for cost estimation.

Our discussion commences with the work of Barry Boehm [Boehm, 2002]. According to the Boehm model for cost estimation, there are five approaches to software cost estimation (more precisely, the software engineering cost estimation) as summarized below:

- **Algorithmic Cost Modeling:** The cost is determined as a function of the effort.
- **Expert Judgment:** A group of experts assess the project and estimate its cost.
- **Analogy:** The project is compared to some other completed project in the same application domain, and its cost is estimated.
- **Parkinson's Law:** The project cost is based on convenience factors such as available resources, and time horizon.
- **Pricing Based on Consumer:** The project is assigned a cost based on the consumer's financial resources, and not necessarily on the software requirements.

Obviously, the latter four proposals are highly subjective and error-prone; they will not be explored any further. The algorithmic approach has generated much interest and subsequent proposals over the past twenty-five years, some of which have been listed for recommended readings.

16.4.1 Algorithmic Cost Models

Algorithmic cost models assume that project cost is a function of other project factors such as size, number of software engineers, and possibly others. As such, each cost model uses a mathematical formula to compute an index for the software engineering effort (E). The cost is then determined based on the

evaluation of the effort. [Figure 16-6](#) outlines the typical cost model.

$$E = A + B * P^C$$

Key:

1. E denotes the effort in programmer-months.
2. A, B and C are empirically derived constants.
3. P is the primary input in LOC or FP.

[Figure 16-6. Typical Cost Model](#)

By way of observation, the exponent C typically lies in the range {0.8 .. 1.5}. The constants A, B, and C are called adjustment parameters, and they are determined by project characteristics (such as complexity, experience of the project team members, the development environment, etc.).

Pressman [Pressman, 2005] lists a number of specific cases-in-point of this cost model, such as:

- $E = 5.2 * (\text{KLOC})^{0.91}$... the Watson-Felix model
- $E = 5.5 + 0.73 * (\text{KLOC})^{1.16}$... the Bailey-Basili model
- $E = 3.2 * (\text{KLOC})^{1.05}$... the COCOMO Basic model
- $E = 5.288 * (\text{KLOC})^{1.047}$... the Doty model for $\text{KLOC} > 9$
- $E = -91.4 + 0.355 * (\text{FP})$... the Albrecht & Gaffney model
- $E = -37 + 0.96 * (\text{FP})$... the Kemerer model
- $E = -12.88 + 0.405 * (\text{FP})$... the small project regression model

Note In the above examples, the acronym KLOC means thousand lines of code.

16.4.2 The COCOMO Model

Boehm first proposed the *Constructive Cost Model* (COCOMO) in 1981, and since then it has matured to the status of international fame. The basic model

was of the form

$$E = B * P^C * (EAF) \text{ where EAF denotes the effort adjustment factor (equal to 1 in the basic model)}$$

Boehm used a three-mode approach, as follows (Figure 16-7 provides the formula used for each model):

- **Organic Mode** — for relatively simple projects.
- **Semi-detached Mode** — for intermediate-level projects with team members of limited experience.
- **Embedded Mode** — for complex projects with rigid constraints.

Organic:	$E = 2.4 * (\text{KLOC})^{1.05}$
Semi-detached:	$E = 3.0 * (\text{KLOC})^{1.12}$
Embedded:	$E = 3.6 * (\text{KLOC})^{1.20}$

Figure 16-7. COCOMO Formulae

The original COCOMO model was designed based on traditional procedural programming in languages such as C, Fortran, etc. Additionally, most (if not all) software engineering projects were pursued based on the waterfall model. The next subsection discusses Boehm's revision of this basic model.

16.4.3 The COCOMO II Model

Software engineering has changed significantly since the basic COCOMO model was first introduced. At the time of introduction, OOM was just an emerging idea, and most software engineering projects followed the waterfall model. In contrast, today, most software engineering projects are pursued in the OO paradigm, and the waterfall model has given way to more flexible, responsive approaches (review [chapter 1](#)). In 1997, Boehm and his colleagues introduced a revised COCOMO II model to facilitate the changes in the software engineering paradigm.

The COCOMO II model is more inclusive, and receptive to OO software development tools. It facilitates assessment based on the following:

- Number of lines of source code
- Number of object-points or function-points
- Number of lines of code reused or generated
- Number of application points

The changes relate to application points and code reuse/generation — features of contemporary OO software development tools. We will address both in what is called the *application composition model*. Two other sub-models of COCOMO II are the *early design model* and the *post-architecture model*.

Application Composition Model

In the application composition model, we are interested in *object points* (OP) as opposed to function points (FP). The model can be explained in seven steps as summarized below:

1. The number of object points in a software system is the weighted estimate of the following:
 - The number of separate screens displayed
 - The number of reports produced
 - The number of components that must be developed to supplement the application
2. Each object instance is classified into one of three complexity levels — simple, medium or difficult — according to the schedule in [Figure 16-8](#).

No. of Views/Sections Contained in a Screen or Report	Number and Source of Data Tables Required		
	Total < 4	Total 5 – 7	Total ≥ 8
< 3	Simple	Simple	Medium
3 – 7	Simple	Medium	Difficult
≥ 8	Medium	Difficult	Difficult

[Figure 16-8. Object Instance Classification Guide](#)

3. The number of screens, reports, and components are

weighted according to the schedule in [figure 16-9](#). The weights actually represent the relative effort required to implement an instance of that complexity level.

Object Classification	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
Component			10

[Figure 16-9](#). Complexity Weights for Object Classifications

- Determine the *object point count* (OPC) by multiplying the original number of object instances by the weighting factor, and summing to obtain the total OPC. [Figure 16-10a](#) clarifies the calculation and [Figure 16-10b](#) provides an illustration.

$$OPC = \sum[(NI) * (W)]$$

Key:

- NI is number of instances of a particular object classification (screen, report, or component) and W is the weight for that object classification.
- OPC is the object point count.

[Figure 16-10a](#). Calculating the OPC

Object Classification	Occurrence	Complexity	Weight	OPC
Screen	10	Simple	1	10
Screen	10	Medium	2	20
Screen	6	Difficult	3	18
Component	6	Difficult	10	60
Total OPC				108

[Figure 16-10b](#). Illustrating Calculation of the OPC

- Calculate the *number of object points* (NOP) by adjusting the OPC based on the level of code reuse in the application:

$$NOP = (OPC) * [(100 - \%Reuse)/100]$$

- Determine a *productivity rate* (PR) for the project, based

on the schedule of [Figure 16-11](#). Note that the schedule that it is in the project's best interest to have a project team of very talented and experienced software engineers, and to use the best software development that is available.

Team Quality	Very Low	Low	Nominal	High	Very High
Development Tool Quality	Very Low	Low	Nominal	High	Very High
Productivity Rate	4	7	13	25	50

Note:

1. Team Quality includes the capabilities and talent of the team members.
2. Development tool quality includes the maturity and capabilities of the software development tool employed.

[Figure 16-11. Productivity Rate Schedule](#)

7. Compute the effort (E) via the formula

$$E = (NOP) * (PR)$$

[Figure 16-12](#) summarizes the steps in the application composition model. With practice on real projects, you will become more comfortable with this cost estimation technique. The important thing to remember about the model is that the software cost is construed to be a function of the engineering effort, and the complexity of the software.

1. Identify screens, reports and components in the software system.
2. Classify each object instance as simple, medium or difficult, based on the schedule of figure 16.8.
3. Assign a complexity weight to each project screen, report, or component according to the schedule of figure 16.9.
4. Calculate the OPC based on the formula $OPC = \sum[(NI) * (W)]$
5. Calculate the NOP based on the formula $NOP = (OPC) * [(100 - \%Reuse)/100]$
6. Determine the PR for the project based on the schedule of figure 16.11.
7. Calculate the effort based on the formula $E = (NOP) * (PR)$

[Figure 16-12. Summary of the Application Composition Model](#)

Early Design Model

The early design model is recommended for the early stages the software engineering project (after the requirements specification has been prepared). A full discussion of the approach will not be conducted here, but a summary follows.

The formula used for calculating effort is

The formula used for calculating effort is

$$E = B * P^C * (EAF) \text{ where EAF denotes the effort adjustment factor}$$

As clarified earlier, B and C are constants, and P denotes the primary input (estimated LOC or FP). Based on Boehm's experiments, A is recommended to be 2.94, and C may vary between 1.1 and 1.24. The EAF is a multiplier that is determined based on the following seven factors (called *cost drivers*). In the interest of clarity, the originally proposed acronyms have been changed:

- Product Reliability and Complexity (PRC)
- Required Reuse (RR)
- Platform Difficulty (PD)
- Personnel Capability (PC)
- Personnel Experience (PE)
- Facilities (F)
- Schedule (S)

$$EAF = (PRC) * (RR) * (PD) * (PC) * (PE) * (F) * S$$

Post-Architecture Model

The post-architecture model is the most detailed of the COCOMO II sub-models. It is recommend for use during actual development, and subsequent maintenance of the software system ([chapter 18](#) discusses software maintenance).

The formula used for calculating effort is of identical form as for the early design model:

$$E = B * P^C * (EAF) \text{ where EAF denotes the effort adjustment factor}$$

In this case, B is empirically recommended to be 2.55 and C is calculated via the formula

$$C = 1.01 + 0.01 \sum(CW) \text{ where } CW \text{ represents capability weights calibrated based on characteristics of the project team and the host organization.}$$

The criteria (also called *scale factors*) used to assign capability weights (CW) about the project and the project team are as follows:

- **Precedence:** Does the organization have prior experience working on a similar project?
- **Development Flexibility:** The degree of flexibility in the software development process.
- **Architecture/Risk Resolution:** How much risk analysis has been done, and what steps have been taken to lessen or eliminate these risks?
- **Team Cohesion:** How cohesive is the team?
- **Process Maturity:** What is the process maturity of the organization? How capable is it in successfully pursuing this project?

[Figure 16-13](#) provides the recommended schedule for determining the capability weights for these criteria. As can be seen from the figure, the weights range from 0 (extra high) to 6 (very low).

Capability Factor	Very Low	Low	Nominal	High	Very High	Extra High
Precedence	4.05	3.24	2.42	1.62	0.81	0.00
Development/Flexibility	6.07	4.86	3.64	2.43	1.21	0.00
Architecture/Risk Resolution	4.22	3.38	2.53	1.69	0.84	0.00
Team Cohesion	4.94	3.95	2.97	1.98	0.99	0.00
Process Maturity	4.54	3.64	2.73	1.82	0.91	0.00

[Figure 16-13](#). Capability Weights Schedule

The EAF is determined from a much wider range of factors than the early design model. Here, there are seventeen factors (cost drivers). [Figure 16-14](#) lists these cost drivers along with their assigned ratings. In the interest of clarity, the original acronyms have been changed. The EAF is the product of these cost multipliers.

Capability Factor	Very Low	Low	Nominal	High	Very High	Extra High
Product:						
Required Software Reliability (RSR)	0.75	0.88	1.00	1.15	1.39	-
Database Size (DBS)	-	0.93	1.00	1.09	1.19	-
Product Complexity (Cplx)	0.70	0.88	1.00	1.15	1.30	1.66
Required Reusability (RR)		0.91	1.00	1.14	1.29	1.49
Documentation (Doc)		0.95	1.00	1.06	1.13	
Platform:						
Execution Time Constraint (ETC)	-	-	1.00	1.11	1.31	1.67
Main Storage Constraint (MSC)	-	-	1.00	1.06	1.21	1.57
Platform Volatility (PV)	-	0.87	1.00	1.15	1.30	-
Personnel:						
Analyst Capability (AC)	1.50	1.22	1.00	0.83	0.67	-
Programmer Capability (ProgC)	1.37	1.16	1.00	0.87	0.74	-
Personnel Continuity (PrsnlC)	1.24	1.10	1.00	0.92	0.84	-
Applications Experience (AE)	1.22	1.10	1.00	0.89	0.81	-
Platform Experience (PE)	1.25	1.12	1.00	0.88	0.81	-
Language and Tool Experience (LTE)	1.22	1.10	1.00	0.91	0.84	
Project:						
Software Tools (ST)	1.24	1.12	1.00	0.86	0.72	-
Multi-site Development MD)	1.25	1.10	1.00	0.92	0.84	0.78
Development Schedule (DS)	1.29	1.10	1.00	1.00	1.00	-

Figure 16-14. Post-Architecture Cost Drivers Schedule

You have no doubt observed that this is quite a complex cost model. It must be emphasized that in order to be of any use, the model must be calibrated to suit the local situation, based on local history. Moreover, there is considerable scope for uncertainty in estimation of values for the various factors. The model must therefore be used as a guideline, not a law cast in stone.

16.5 Summary and Concluding Remarks

Let us summarize what we have discussed in this chapter:

- Software cost is comprised of equipment cost, facilities cost, engineering cost and operational cost. Equipment cost, facilities cost, and operational cost, are determined by following standard business procedures. Engineering cost may be determined using standard business procedures also,

but in software engineering, we are also interested in fine-tuning the estimation of this cost component by exploring more deterministic models.

- Software price is influenced by factors including (but not confined to) software cost, market opportunity, uniqueness, usefulness and quality, contracting terms, requirements volatility, and financial health of the organization that owns the product.
- Software value is influenced by software cost, productivity brought to the organization, and cost savings brought to the organization.
- Software productivity may be evaluated based on the software engineering effort employed in planning and constructing the product, or based on the value added to the organization. Two metrics used for evaluating the engineering effort are the size-based metrics, and the function-based metrics. Two methods for evaluating value added are the difference method, and the NPV analysis method.
- The size-based metrics estimate productivity based on the number of lines of code and pages of documentation of the software.
- Function-based metrics estimate productivity based on the number of function-points of the software.
- Value-added metrics attempt to estimate the value added to the organization by the software.
- Techniques for estimating engineering costs include algorithmic cost modeling, expert judgment, analogy, Parkinson's Law, and pricing based on consumer. Algorithmic cost modeling presents much research interest in software engineering.
 - The typical formula for evaluating engineering effort in algorithmic cost modeling is $E = A + B * P^C$.
 - The COCOMO model uses three derivations of the basic algorithmic cost modeling formula. The model relates to

traditional systems developed in the FO paradigm.

- The COCOMO II model is a revision of the basic COCOMO model, to facilitate more contemporary software systems developed in the OO paradigm. It includes an application component model, an early design model, and a post-architecture model.
- The application composition model outlines a seven-step approach for obtaining an evaluation of the engineering effort of the software system. This is summarized in [Figure 16-12](#).
- The early design model uses an adjusted formula for evaluating the engineering effort. The formula is $E = B * P^C * (EAF)$, and certain precautions must be followed when using it.
- The post-architecture model uses the same formula $E = B * P^C * (EAF)$; however, the precautions to be followed are much more elaborate.

We have covered a lot of ground towards building software systems of a high quality. The deliverable that comes out of the software development phase is the actual product! It is therefore time to discuss implementation and maintenance. The next three chapters will do that.

16.6 Review Questions

1. What is the difference between software cost and software price?
2. What are the components of software cost? What are the challenges to determining software cost?
3. State and briefly discuss the main factors that influence software price?
4. How is software value determined? What are the challenges to determining software value?

5. State three approaches to evaluating software productivity. For each approach, outline a methodology, and briefly highlight its limitations.
6. Identify Boehm's five approaches to software cost estimation. Which approach provides the most interest for software engineers?
7. Describe the basic algorithmic cost model that many software costing techniques employ.
8. Describe the COCOMO II model.
9. Choose a software engineering project that you are familiar with.
 - a. Using the COCOMO II Application Composition Model, determine an evaluation of the engineering effort of the project.
 - b. Using the COCOMO II Early Design Model, determine an evaluation of the engineering effort of the project.
 - c. Using the COCOMO II Post-Architecture Model, determine an evaluation of the engineering effort of the project.
 - d. Compare the results obtained in each case.

16.7 References and/or Recommended Reading

[Albrecht, 1979] Albrecht, A. J. "Measuring Application-Development Productivity." *AHARE/GUIDE IBM Application Development Symposium*. 1979. See chapter 26.

[Boehm, 1981] Boehm, BarryW. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[Boehm et. al., 1997] Boehm, Barry W., C. Abts, B. Clark, and S. Devnani-

Chulani. COCOMO II Model Definition Manual. Los Angeles, CA: University of Southern California, 1997.

[Boehm, 2002] Boehm, Barry W., et. al. COCOMO II.
<http://sunset.usc.edu/research/COCOMOII> (accessed July 2006).

[Fenton, 1997] Fenton, Norman E. and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Boston, MA: PWS Publishing, 1997.

[Humphrey, 1990] Humphrey, Watts S. *Managing the Software Process*. Reading, MA: Addison-Wesley Publishing, 1990.

[Jones, 1997] Jones, Capers. *Applied Software Measurement: Assuring Productivity and Quality* 2nd ed. New York, NY: McGraw-Hill, 1997.

[MacDonell, 1994] MacDonell, Stephen G. “Comparative Review of Functional Complexity Assessment Methods for Effort Estimation.” BCS/IEE Software Engineering Journal 9(3), 1994. pp.107–116.

[Pressman, 2005] Pressman, Roger. *Software Engineering: A Practitioner’s Approach* 6th ed. Crawfordsville, IN: McGraw-Hill, 2005. See chapter 23.

[Putnam, 1992] Putnam, Lawrence H. and Ware Myers. *Measures for Excellence: Reliable Software on time, within Budget*. Englewood Cliffs, NJ: Yourdon Press, 1992.

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See chapter 26.

PART E



Software Implementation and Management

This division addresses the implementation and management of computer software. The objectives of this division are as follows:

- To emphasize the importance of having a wise software implementation plan
- To underscore the importance of good software management as an essential aspect of good software engineering
- To discuss the alternatives for organizing software engineering enterprises and/or ventures

The division consists of the following chapters:

- [Chapter 17](#) — Software Implementation Issues
- [Chapter 18](#) — Software Management
- [Chapter 19](#) — Organizing for Effective Management

CHAPTER 17



Software Implementation Issues

Having successfully engineered the software product, it must be implemented in an environment where end users will find it useful. Unless this is done, the whole effort involved in investigation, analysis, design and development of the product would have been pointless. This chapter discusses important software implementation issues under the following subheadings:

- Introduction
- Operating Environment
- Installation of the System
- Code Conversion
- Change Over
- Training
- Marketing of the Software
- Summary and Concluding Remarks

17.1 Introduction

Irrespective of how the software is acquired (review [chapter 1](#)), it must eventually be implemented. Planning and preparation for this system implementation should start long before the completion of the acquisition. If the implementation is not carefully planned and all factors considered, the exercise can be very frustrating and misrepresenting of the system and professionals responsible for its introduction.

Poor implementation can cause the failure and rejection of a well-designed software system that actually meets the needs of its intended users. This underscores that software engineering does not end after product development. Users must be trained to use the product. To this end, the system must be installed, configured and monitored. If your organization is in the business of marketing computer software, then part of the implementation would be the development and pursuit of a marketing plan for the product. These and other related matters will be addressed in the upcoming sections.

17.2 Operating Environment

Consideration about the operating environment addresses the following questions:

- Will the system be centrally operated, or will a distributed computing environment be in place?
- What precautions will be necessary, given the environmental constraints?

17.2.1 Central System

The centralized approach is the traditional approach, where the software system runs on a particular machine, and is inaccessible except through that machine. Traditionally, data processing (DP) departments used this approach to manage centralized information systems that provided information services for the other departments in the organization. All data entry, maintenance, and processing were done centrally and reports were sent to various departments. This approach was (and still is) particularly useful in a batch-processing environment.

Advantages of this approach are as follows:

- There is a central locus of control, which allows for easy tracking of system problems.
- There is little or no ambiguity about accountability.
- The approach is ideal for embedded systems that do not need to interact with multiple end users.

The approach suffers from the following disadvantages:

- For information systems, the likelihood of interdepartmental delays regarding the transfer of information is very high. Information may therefore not arrive at the computing center on time. This would trigger a late entry of data into the system, which in turn would cause the late generation of reports. This lateness factor could ripple through the entire organization, causing untold problems.
- The organization's information system would not be very responsive.
- The approach is very restrictive and inflexible.

17.2.2 Distributed System

The distributed approach is the contemporary approach to software system implementation. All users have access to the software system (via workstations). Data enters the system at various points of origin — via workstations operated by users in different departments, or via electronic transfers (that may be transparent to end users). The software system is accessed by various end-users as required without any intervention from the software engineering team.

Two broad approaches can be identified:

1. Distributed workstations are connected to a central network server in a local area network (LAN), a wide area network (WAN), or a metropolitan area network (MAN).
2. A distributed network (LAN, WAN or MAN), consisting of a conglomeration of servers and workstations, and encompassing different departments and/or branches, is constructed and managed.

There may be various configurations (topologies) of each approach, but this is a subject for another course (in either *electronic communication systems* or *computer networks*).

Advantages of the distributed approach include the following:

- The approach is ideal for multi-user environments.

- For information systems, immediate access to the system (especially mission critical information) is provided.
- For information systems, accountability on data accuracy is shifted from the software engineering team to user departments.
- The software engineering team can concentrate on ensuring that the system provides users with appropriate interfaces, data validity, system performance and other technical issues.
- Through user training and interaction, a clearer understanding between end-users and software engineering team is enhanced.

Disadvantages of the approach include the following:

- User training can be challenging. For information systems, even after comprehensive training, the odd user may key in inaccurate data and try to blame “the system” or the “IS/IT Department.”
- In the absence of adequately trained personnel, this approach could be a prescription for chaos.

17.2.3 Other Environmental Issues

Other environmental issues to be addressed in software implementation include (review [chapter 3](#))

- The availability of adequate power supply
- Cooling (or heating) requirements
- Physical security and accessibility requirements

17.3 Installation of the System

In preparation for system installation, a fundamental question to address is whether the installation will involve new machines or just software.

If new machines are involved, an installation diagram (plan) showing where certain equipment will go, is required. Consideration must be given to the various installation alternatives (for example, in the case of a computer network various topologies apply) and the most appropriate one chosen.

If installation involves software only, then consideration should be given to

- The effect on the existing system
- Precautions involved
- Sites of installation (in case of a distributed system)

If installation involves both hardware and software, then obviously, considerations must be given to both areas as outlined above.

17.4 Code Conversion

Code conversion is applicable where a software system replaces an existing one, and the coding system used to identify data in one system is different in the other. In such a circumstance, the software engineer must do the following:

- Analyze both systems and clearly identify points of differences in the coding systems.
- Design and test a methodology of linking the differing coding systems.
- Design and test a methodology for converting data from the old format to the new format.

In many cases, some amount of interface coding is necessary. [Figure 17-1](#) represents the main components of a code conversion system, while [Figure 17-2](#) provides an example. The interface program(s) provide(s) conversion and communication between old and new codes.

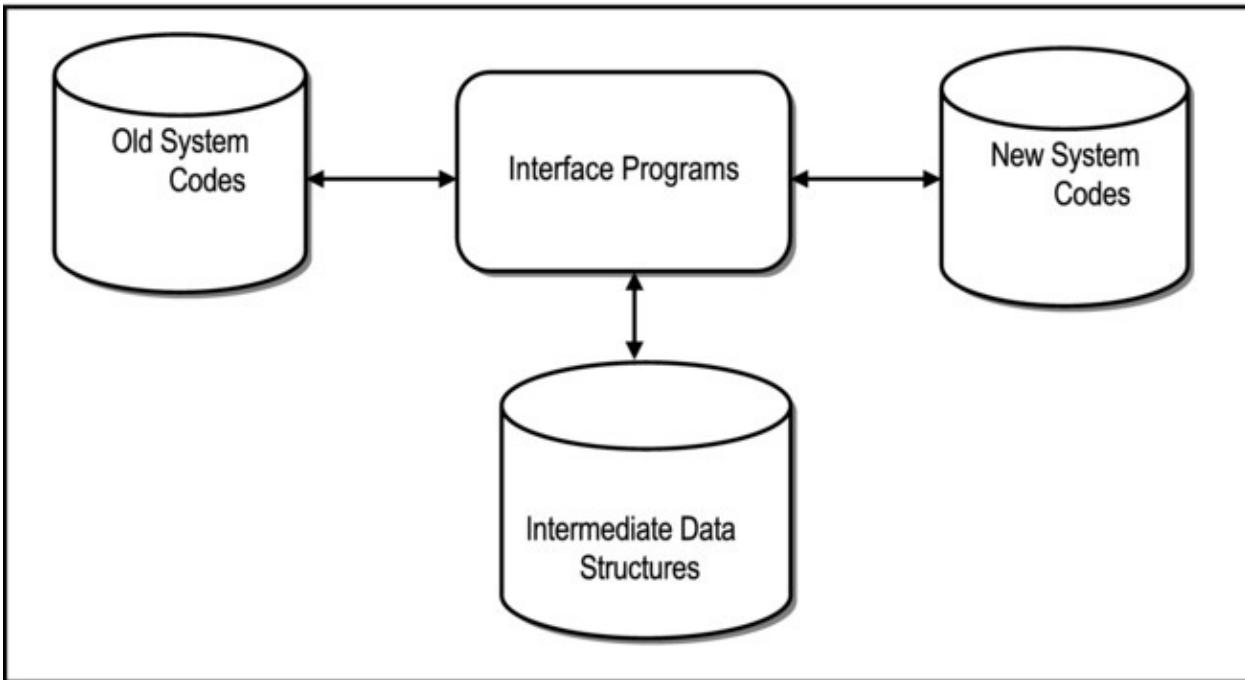


Figure 17-1. Components of a Code-Conversion System

Suppose that an existing information system that used an alphanumeric code (A5, say) for account number is being replaced by a new information system that uses a numeric code (N9, say) for account numbers. Suppose that there is no logical relationship between old codes and new codes. However, it is desired to link new accounts of the new system with their old counterparts in the old system (the old system might contain a lot of historical but needed data which will eventually be converted).

The intermediate database table (**OldNewLink**, say) could be designed to store both old and new codes as follows:

Old Code (A5)	New Code (N9)
AB001	000450000
...	...
XY999	099590001

Each **old code** value is assigned a **new code** value. Intermediate ADD and UPDATE operations would allow for maintenance of the table. Another intermediate RETRIEVAL operation would retrieve a **new code** value, given an **old code** value and vice versa.

Figure 17-2. Code Conversion Example

17.5 Change Over

The previous section provided a clue to the current section: change over from an old software system to a new one. Four strategies can be identified:

- Direct changeover
- Parallel conversion
- Phased conversion
- Distributed conversion

17.5.1 Direct Change Over

The essence of direct changeover approach is that on a given date, the old system is dropped and the new system is put into use. After changeover, users cannot use the old system.

This approach can only be pursued if the new system (with all interfaces) is thoroughly tested and shown to be completely acceptable. At best, a minimal delay is expected.

17.5.2 Parallel Conversion

In the parallel conversion approach, the new and old systems are simultaneously run until users are satisfied with the new system. Then and only then is the old system discarded. This is traditionally the strategy used in converting from manual to computerized systems, but it may also be applied to changeover from an old computerized system to a newer one.

The approach gives users a sense of security, but is costly in terms of effort. This is so because in some cases, two sets of workers have to be employed to work on the parallel systems; in others cases, employees may be called upon to work overtime.

17.5.3 Phased Conversion

In phased conversion, change from old to new is gradual over a period of time. Aspects of the new system are gradually introduced. The approach is consistent with the phased prototype model of [Chapter 1](#), but also applies to situations where the system was acquired by other methods apart from direct development.

The main drawback in phased conversion can be protracted over an extended period of time. This could potentially create anxiety between the project team and end users.

and end users.

One advantage is that each component is thoroughly tested before use. Also, users are familiar with each component before it becomes operational. The main drawback is that of interfacing among components.

17.5.4 Distributed Conversion

In the distributed conversion approach, many installations of the same system are contemplated. Based on the evaluations of these installations a decision is made with respect to global implementation.

The approach is very common in large financial institutions with several branch offices (e.g. introducing ATM machines at various sites, a bank take-over, etc.).

17.6 Training

Training of end users is an integral part of software engineering. The training strategies are often determined by the following issues:

- Who is being trained
- Who is conducting the training
- What resources are available for the training
- Where the training will be conducted

Training may be conducted by software engineers, vendors, or in-house users of the software system. The following table ([Figure 17-3](#)) provides a rough guideline as to who may conduct training, given certain circumstances.

Scenario	Recommended Trainer
Principal users being trained for the first time	Software Engineer or qualified Vendor
Secondary users who are power players	Software Engineer
In-house (review) training of principal users	Specially trained in-house users or Vendor
New marketing opportunity	Software Engineer or specially trained Marketer

[Figure 17-3](#). Who Should Train?

Training may be conducted at a specific (and specially prepared) training

center, or on-site (at the request of the organization or department receiving the training), the latter being more expensive.

Training objectives (and performance criteria) must be clearly communicated to trainees as this helps them to appreciate what is expected of them, and to adequately prepare for the exercise. Who is being trained will affect the training objectives. For example, clerical staff for data entry requires a different approach from managers of an organization.

Methods used in training should involve visual, hearing, and practical (hands-on training) as is appropriate.

Training materials must be well prepared. Training materials are usually in the form of user guides, summary sheets and workbooks.

If trainee evaluation is required, it must be objective and based on the established performance criteria set. In some instances, the instructor may also be evaluated by the trainees, as a means of quality assurance by the company (or department) offering the training.

17.7 Marketing of the Software

If the product was developed by a software engineering company, to be placed on the market as a consumer product, then the earlier mentioned issues would be part of a much larger marketing strategy.

Issues such as operating environment, installation and code conversion, would be addressed in the product documentation. Changeover would be left up to the purchasing consumer. Training might be handled by the company, or other vendors who might be marketing the product.

Contemporary marketing strategies are typically based on the following five principles:

- **Product:** This relates to the product line(s) the software will be marketed with/as.
- **Pricing:** This relates to how the software will be priced (review [Chapter 16](#)).
- **Promotion:** This includes both *positioning* and *packaging* of the software. Positioning relates to how the product is introduced and marketed in the market place. Packaging

relates to what other products that are marketed with the software.

- **Placement (or Physical Distribution):** This relates to the avenues through which the product eventually gets to the consumer.
- **People:** This relates to the customer services (including training and support) that will be provided for the product.

The marketing strategy is typically guided by a market research, which is planned and conducted by suitably qualified individuals. The findings of the research are then used to drive the marketing strategy.

17.8 Summary and Concluding Remarks

Here is a summary of what we have discussed in this chapter:

- Planning the software implementation is critical to the user acceptance of the software system.
- The operating environment for the software product may be a central system, or a distributed system.
- System installation must be carefully planned.
- If the software system is replacing a pre-existing system, then the matters of code conversion and system changeover become very important.
- System changeover may be direct, parallel, phased, or distributed.
- Implementation often requires training of the end users. This must be carefully planned.
- If the software system is to be marketed to the consuming public, then a marketing plan for the product must be developed and followed.

The following deliverables should be available at the end of a software engineering project:

- Initial System Requirements
- Feasibility Analysis Report
- Requirements Specification
- Design Specification
- System Manual(s)
- User's Guide(s)
- Help System
- The operational Software Product

The requirement specification, design specification, system manual(s), user guide(s) and help system constitute the software documentation. These items must be maintained along with the operational product. The next chapter will discuss software maintenance.

17.9 Review Questions

1. What are the critical issues to be managed during the software implementation?
2. If you were in charge of outfitting your college or university with a strategic information system for its core functions:
 - What operating environment would you consider? Why?
 - Describe a plan for the training of the faculty and staff.
 - What strategy would you use for transitioning from an archaic system on which the institution relied heavily, to your new system? What precautions would you take?

3. Discuss the importance and relevance of code conversion during software implementation.
4. What are the possible approaches to system changeover? When would you use each approach?
5. What guidelines would you follow in conducting user training for a new software system?

17.10 Recommended Readings

[Kendall, 2005] Kendall, Kenneth E. and Julia E. Kendall. *Systems Analysis and Design* 6th ed. Upper Saddle River, NJ: Prentice Hall, 2005. See [Chapter 17](#).

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [Chapters 10](#) and [11](#).

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [Chapter 14](#).

CHAPTER 18



Software Management

This chapter discusses software management as a vital part of the software engineering experience. As you are aware, it is the final phase of the SDLC. However, the truth is, unless you are very lucky, it is the most likely entry point into a career as a software engineer: In the marketplace, you will most likely have to start off by carrying out management responsibilities on existing software before you get a chance to develop a complete system from scratch. Being part of a project team that develops a major system from scratch is an achievement that every software engineer should aspire to experience at least once in his/her career. However, it is probably worth remembering that many practicing software engineers never experience this.

Software management involves product maintenance, product integration and product reengineering. Against this background, the chapter proceeds under the following captions:

- Introduction
- Software Maintenance
- Legacy Systems
- Software Integration
- Software Re-engineering
- Summary and Concluding Remarks

18.1 Introduction

Once the software has been implemented, it then enters a usage phase. During

this time, the requirements of the software may change. Software maintenance ensures that the software remains relevant for the duration of its life cycle. Failure to maintain the software can reduce its life cycle, thus making it irrelevant and/or unwanted.

If, due to design and/or development flaws, or obsolescence, it has been determined that the cost of maintaining the software (nearly) outweighs the cost of replacing it, then at such point, a decision needs to be taken about the future role and usefulness of the software as well as its possible replacement. However, in the absence of such critical circumstance, the software must be maintained.

As you proceed through this chapter, bear in mind that various factors will affect the management decisions taken about a software product. Three such factors are mentioned in [Figure 18-1](#).

Size & Complexity: As the size and complexity of the software product increases, the likelihood of having a proliferation of such type (category) of software decreases, and the likelihood that companies involved in such projects will seek to maintain their existing products increases.

Technology: If a product is based on obsolete technology, its parent company will eventually be faced the prospects of discontinuing marketing of the product, replacing the product, or reengineering the product.

Support: The maintenance provided is dependent on the availability technical, financial, technological and human resources.

Figure 18-1. Factors Affecting Software Management Decisions

18.2 Software Maintenance

In order to ensure that the software product remains relevant, despite the changing requirements of the user environment(s), software maintenance is necessary. We shall examine software maintenance under three subheadings:

- Software Modifications
- Software Upgrades and Patches
- Maintenance Cost

18.2.1 Software Modifications

Software modifications may be put into three broad classifications:

- Enhancements
- Adaptive changes
- Corrective changes

Enhancements are modifications that add to the usefulness of the software. Enhancements may add/improve features, add/improve functionality, or broaden the scope of the software. Most software changes tend to be enhancements.

Adaptive changes are peripheral changes, caused mainly due to changes in the operational environment of the software.

Corrective changes address flaws (bugs) in the software; a well-designed software product will have few bugs. Poorly designed software systems tend to have more bugs than well-designed ones.

Software modifications are not to be made in a flippant manner, but in a manner that ensures that established standards are conformed with. Typically the software manager establishes a formal organizational procedure for effecting software change. It may include activities such as

- Request by user on a standard change-request form (as illustrated in [Figure 18-2](#))

Request Date:	<input type="text"/>
Requesting Department:	<input type="text"/>
Department Representative:	<input type="text"/>
Existing Software System:	<input type="text"/>
State the observed system limitation: <input type="text"/>	
2. State your required system modification: <input type="text"/>	
3. Explain how this change will affect your work: <input type="text"/>	

Figure 18-2. System Modification Request

- Evaluation of the request by a user liaison officer and/or the software manager
- Evaluation of the request by a software engineer
- Decision taken on the request
- Action taken in response to the request

18.2.2 Software Upgrades and Patches

A software *upgrade* is a modification to a software product to improve its functionality or performance in specific areas. The upgrade may be in any combination of the following possibilities:

- Addition of new features (and possibly program files) that seamlessly integrate into the existing software
- Improvement of existing features and/or functionalities by replacement of existing program files with revised program files
- Removal of known bugs by replacement of existing program files with revised program files

Software engineering companies that market products to the consuming public usually issue software upgrades in the form of releases. Traditionally, software releases are typically numbered in the format that the sections of this text are numbered. However, a more recent trend is to number the versions based on the year of release and to add descriptive qualities to the version names. Following are three examples in three different product lines:

- Early versions of Borland Delphi were of the form V1Rx (or V1.x) up to V8Rx (or V8.x). This was followed by releases of Delphi 2006 Professional, Delphi 2006 Architect, and Delphi 2006 Enterprise; this trend continued into 2010 versions of the product line (except that it is now marketed by Embarcadero, and not Borland). Since then, the version names have been Delphi XE, Delphi XE2, Delphi XE3, and Delphi XE4.
- Microsoft product lines have been consistently named based on the year of introduction.
- Companies such as Oracle and IBM tend to favor the more traditional approach along with descriptive names, but Oracle applies a slight deviation, using version names such as Oracle 9I, Oracle 10G, Oracle 11G, and Oracle 12C.

If you are working for an organization that does not market its software systems, this meticulous version naming convention may not be necessary, but would nonetheless be advantageous.

Software upgrades from these established software engineering companies

are usually made available via common portable storage media as well as via downloads from the World Wide Web (WWW). Again, if your organization is not in the business of marketing its software products, then this approach may or may not be necessary, depending on the prevailing circumstances.

A software *patch* is a modification to a software product to improve its functionality or performance in specific areas. The patch is more narrowly focused than an upgrade; it may be in any combination of the following possibilities:

- Improvement of existing features and/or functionalities by replacement of existing program files with revised program files
- Removal of known bugs by replacement of existing program files with revised program files

Software engineering companies that market products to the consuming public usually issue software patches in the form of service patches (SP). They are typically numbered sequentially (example SP1, SP2, etc.), with clear indications to consumers as to what versions (releases) of the software product, the patches relate to. Patches are commonly made available via the WWW, but may also be placed on common portable storage media.

18.2.3 Maintenance Cost

The more poorly designed the software, the more maintenance will be required and the higher will the maintenance cost be. Conversely, the better the design, the lower the maintenance cost is likely to be.

Maintenance cost is influenced by the following factors:

- **Lifetime of the software:** In many (but not all) cases, longer lifetime means more maintenance.
- **Quality of the software design:** Generally speaking, poor design leads to high maintenance cost.
- **Amount of changes required:** The greater the number of changes required, the higher the maintenance cost is likely to be.

- **Quality and stability of the support staff:** A very skilled and stable support staff is likely to reduce the maintenance cost.
- **Complexity of software design and configuration:** The more complex the design, the higher the maintenance cost is likely to be (there may be exceptions to this).
- **The development software used** (programming language, DBMS, etc.): Maintenance cost may be constrained by the cost of the software development tools employed.
- **Software size** (in terms of components and source code): Generally speaking, bigger the software system, the higher the maintenance cost (there may be exceptions to this).

As software engineering becomes more standardized, the cost of software maintenance should progressively lessen. If the maintenance cost is going in the opposite direction as time increases, this is a certain alert signal to the organization to be prepared to replace the product at some point in the future.

18.3 Legacy Systems

A *legacy system* is a system that was constructed using relatively old technology, but its useful life continues within the organization. Companies have legacy systems due to a number of compelling reasons:

1. A company, having made huge investments in a system, is not prepared throw it out, since this would mean significant financial loss.
2. The system might be very large and/or complex. Replacing it would therefore be very costly.
3. The system is mission critical: the company's life revolves around it.
4. The complete system specification may not be known or documented, so that replacing it could be risky.
5. Business rules may be embedded (hidden) in the software

and may not be documented anywhere else. Attempting to replace the software would therefore be risky.

6. Any attempt to replace the system must also include the conduct of a data conversion process that offers (infinitesimally close to) 100% guaranteed success. Companies prefer to delay this ultimate activity until they have no alternative, and/or they are guaranteed success.

There are three alternatives for treating legacy systems:

- Continue maintaining the system. If the system is stable and there are not many user requests for changes, this is the most prudent alternative.
- Scrap the old system and replace it with a new one. This alternative should be pursued when the following conditions hold: Firstly, the old system is showing signs of being a liability rather than an asset, that is, it has (almost) reached its scope of usefulness. Secondly, it has been determined that further maintenance would be an effort in futility. Finally, the company is confident that it has gathered enough information to successfully redesign the system.
- Transform the system to improve its maintainability. This process is referred to as software re-engineering, and will be discussed shortly.

18.4 Software Integration

In many cases, and for a number of reasons, a software product may be excellent in the provision of certain desirable services but deficient in the provision of others. In other cases, it might be difficult to find a single software product that provides all the services that an organization is looking for. Should such organizations abandon the investments made in these individual products, in search of a single product? Not necessarily. Rather, the organization could explore the path of software integration.

In software integration, a number of component software products are made to peacefully coexist – i.e. they are integrated. This is consistent with the CBSE

model of earlier chapters (review [chapters 1](#) and [9](#)). In order to provide them with the set of services that they require, large organizations such as banks, insurance companies and universities often use a conglomeration of software products, merged together in an integrated *information infrastructure*. The term *information infrastructure* is used to mean the complete environment (hardware and software) of the organization, and is more appropriately discussed in a course in *strategic information management*.

Software integration has become a very important aspect of software engineering. In recognition of this, large software engineering companies (for example Microsoft, IBM, Oracle, Borland, etc.) typically have the services of integration experts who are specially trained to provide technical advice and expertise to organizations that need to integrate component software products. Evidently, these experts are trained to promote the bias of the companies they represent. However, with some research (and by asking pointed questions), you can obtain a good perspective of what your integration options are. In many cases, the component products may be using different software standards. If this is the case, then the integration team must be prepared to write some interface coding.

Software integration has become a commonplace in contemporary software engineering. As more software products are written to ubiquitous standards, we can anticipate further proliferation of integration options. This is good for the industry, as it will force higher quality software products.

18.5 Software Re-engineering

In a situation where the system quality continues to deteriorate, and the level of user request for changes, as well as the cost of maintenance continues to be high, software re-engineering is the preferred treatment of legacy systems.

Re-engineering may include the following:

- Revising the underlying database of the system (a huge undertaking)
- Using a more modern or sophisticated development software (CASE tool, DBMS suite, or software development kit)
- Refining and/or revising business processes in the organization

- Superimposing a new user interface (wrapper) on top of an existing database

The main activities of software re-engineering are as follows:

- Source code translation or replacement
- Database transformation or replacement (a huge undertaking)
- Reverse engineering (i.e. the existing software is analyzed and the information extracted is used to document its organizational and functional requirements)
- Modernization (related components are grouped together, and redundancies removed; this may involve architectural transformations)
- Data conversion (from old format(s) to new format(s))

Software re-engineering is quite an involved and complex exercise, and must be carried out by qualified and experienced software engineers.

18.6 Summary and Concluding Remarks

It is time once again to summarize what we have covered in this chapter:

- During the usage phase of a software system, it must be managed. Factors affecting effective management of the software include size and complexity, technology, and support.
- Software maintenance involves modifications, upgrade, and the management of maintenance cost. Software modifications may be in the form of enhancements, adaptive changes, or corrective changes. Software upgrades come in the form of releases and service patches. The maintenance cost varies inversely to the software quality.

- A legacy system is a system that was constructed using relatively old technology, but its useful life continues within the organization. The software engineer must be familiar with the different ways of treating legacy systems.
- Software integration is the act of merging different software components so that they peacefully coexist. Component based software engineering (CBSE) has become very popular in recent years.
- Software re-engineering is useful in situations where the system quality continues to deteriorate, and the level of user request for changes, as well as the cost of maintenance continues to be high.

Effective management of a software system can lengthen its useful life, and thereby increase the organization's return on its investment.

18.7 Review Questions

1. Describe how factors such as complexity, technology and support affect decisions taken about the maintenance of a software product.
2. Consider four categories of software. For each category, discuss how the above-mentioned factors (complexity, technology and support) have an influence on the maintenance programs for software products in each respective category.
3. Describe three types of changes that are likely to be made to a software product during its useful life.
4. What is the difference between a software upgrade and a software patch? Using appropriate examples, explain how engineering companies typically handle software upgrades and patches.
5. State four factors that affect the cost of software maintenance.

6. What are legacy systems? Why do they abound and will likely continue to do so for the foreseeable future?
Describe three approaches for dealing with legacy systems.
7. When should software integration be considered? What benefits are likely to accrue from software integration?
8. When should software reengineering be considered?
Describe the main activities that are involved in the reengineering process.

18.8 References and/or Recommended Readings

[Pfleeger, 2006] Pfleeger, Shari Lawrence. *Software Engineering Theory and Practice* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006. See [Chapter 11](#).

[Pressman, 2005] Pressman, Roger. *Software Engineering: A Practitioner's Approach* 6th ed. Crawfordsville, IN: McGraw-Hill, 2005. See chapter 27.

[Schach, 2005] Schach, Stephen R. *Object-Oriented and Classical Software Engineering* 6th ed. Boston, MA: McGraw-Hill, 2005. See [Chapter 15](#).

[Sommerville, 2006] Sommerville, Ian. *Software Engineering* 8th ed. Reading, MA: Addison-Wesley, 2006. See chapters 27 and 28.

CHAPTER 19



Organizing for Effective Management

The final topic in this introductory course in software engineering will address the matter of organizing for effective work. The issue was first introduced in [chapter 1](#), and revisited in [chapters 2](#) and [3](#). This chapter revisits the matter once more, this time focusing on the organizational structure that must be in place in order to support and facilitate good software engineering.

Let us for the moment concentrate on non-software engineering organizations: Some of these organizations have Information Systems (IS) departments/divisions; others have Information Technology (IT) departments/divisions; others have Software Engineering (SE) departments/divisions. Generally speaking, IT is regarded as the broader term, and when used, often includes IS or SE in its scope. However, in many circumstances, IS/SE is used loosely to include IT functions as well. Whether a division or a department is in place is to a large extent, a function of the size of the organization. Large organizations tend to favor IT, SE, or IS divisions that are in turn made up of two or more departments; each department may consist of two or more units or sections. Smaller organizations tend to have IT, SE, or IS departments that may consist of smaller units or sections. Whether a division or a department is in place, there is usually a top IT/SE/IS professional who is ultimately in charge of all the IT related operations. This individual typically operates under the job title of Director or Chief Information Officer (CIO). It is imperative that the appropriate authority and scope of control be accorded to the CIO. In many cases, this translates to the incumbent reporting to the President or the Chief Executive Officer (CEO). The scenario where the CIO reports to the Chief Financial Officer (CFO), though prevalent in many smaller, more traditional organizations, is hardly tenable.

In software engineering firms, the approach is somewhat different. Not only do these organizations need IT support, they are in business to provide IT/SE services to the public.

This chapter examines both scenarios. The discussion proceeds under the following captions:

- Introduction
- Functional Organization
- Parallel Organization
- Hybrid Organization
- Organization of Software Engineering Firms
- Summary and Concluding Remarks

19.1 Introduction

[Chapter 3](#) alluded to the alternatives to organizing a software engineering team, without much discussion. We shall revisit this issue here, and add more clarity. How the IT, SE, or IS division/department is organized will vary from one organization to the other, depending on the following factors:

- The role that IT plays in the organization
- The size of the organization
- The complexity and scope of the organization's information infrastructure
- The nature of the business
- The preference of the CIO

Three approaches to organizing the IT, SE, or IS division/department have been observed:

- The functional approach
- The parallel (project-oriented) approach
- The hybrid approach

Since these approaches apply mainly to non-software engineering companies, attention will also be given to how software engineering firms are

typically organized.

19.2 Functional Organization

The functional approach is the most stable and widely used approach. Sections (meaning departments of a division or units of a department) are defined to reflect a specialization (division) of labor. [Figure 19-1](#) illustrates this structure.

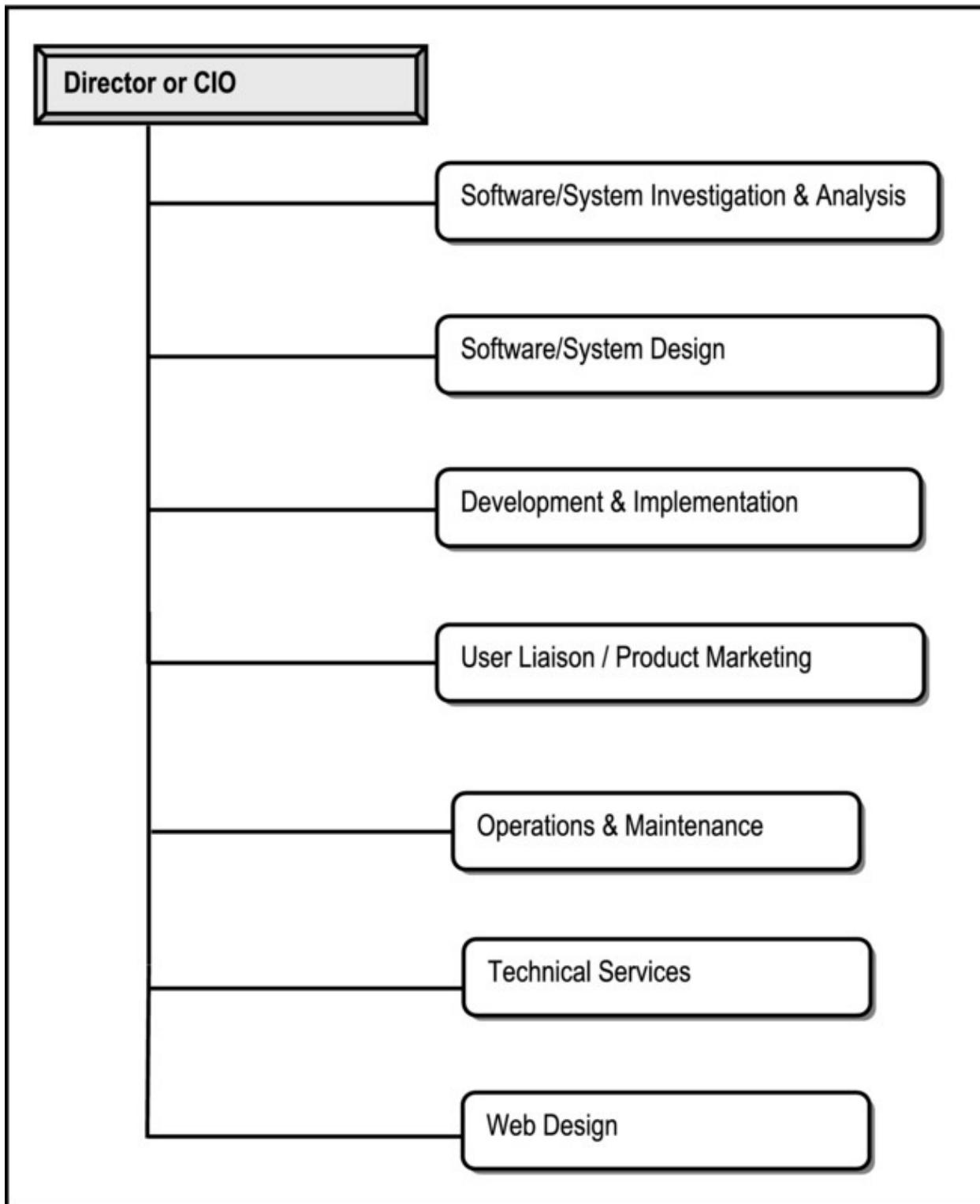


Figure 19-1. Functional Organization

Advantages of this approach are as follows:

- There are well-defined areas of specialization; employees in these areas will be very proficient at their work.
- The reduced span of control promotes effective communication within the functional sections.
- The approach forces good product documentation.
- The software product created is likely to be one of high quality.
- It provides an orderly approach to achievement of long-term goals.
- The approach provides a sense of stability to team members; for instance, people like the idea of having clearly defined job titles, offices, and roles.

Disadvantages of this approach are as follows:

- Inter-group communication may be strained.
- Because a single project is spread over several sections, it might be more difficult to meet targets.
- The approach promotes a lack of overall perspective of a given project, or the broader information infrastructure among employees of any section. Having job rotations can significantly minimize this drawback.

This approach is ideal for a medium/small organization with a limited (manageable) number of projects in a given time horizon. However, it is widely used in small, medium, and large organizations due to its stability.

19.3 Parallel Organization

In the parallel organization (also called project-oriented) approach, the division/department is split based on specific projects. The idea is to preserve the coherence of software engineering projects. [Figure 19-2](#) illustrates this structure.

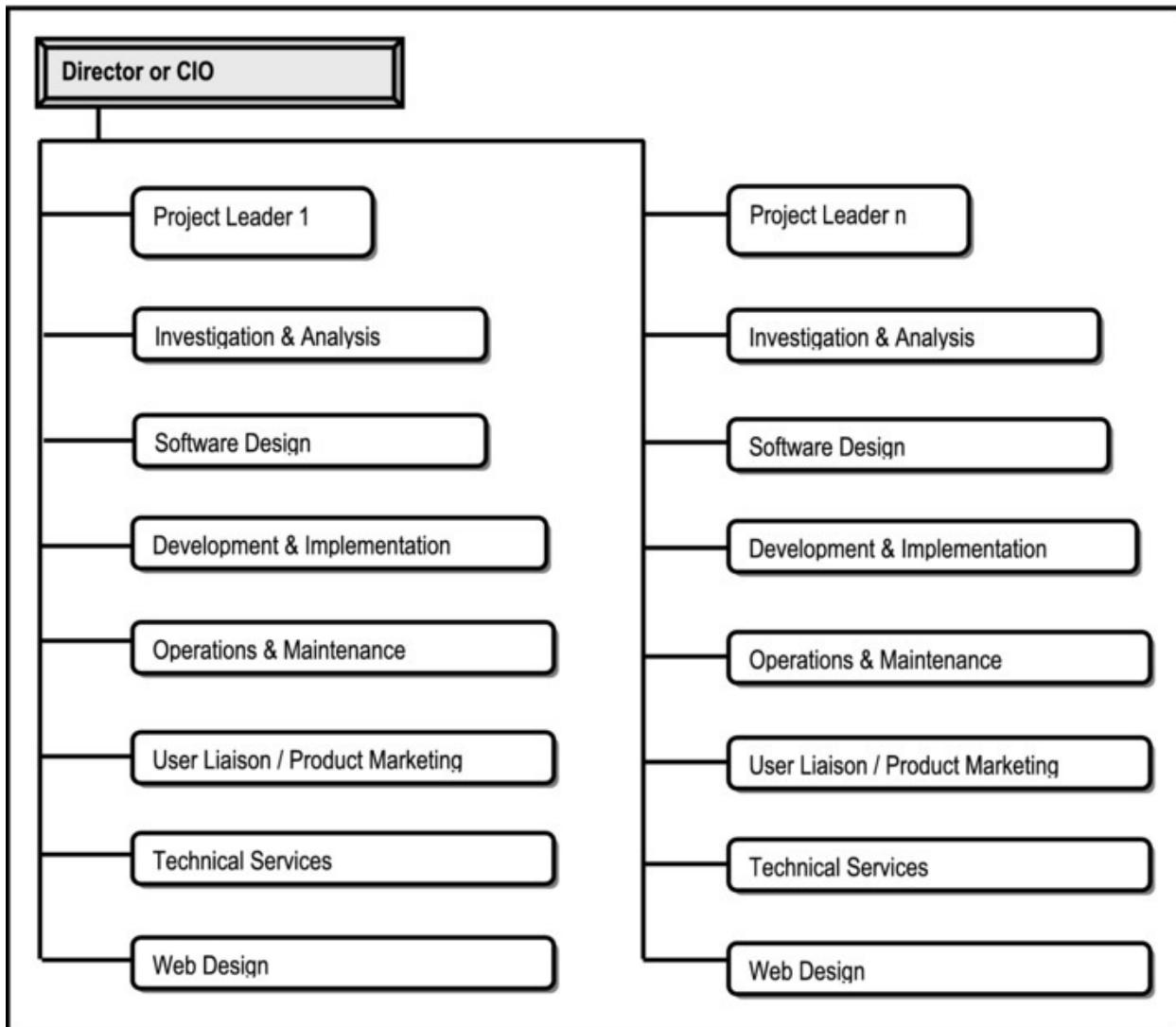


Figure 19-2. Project-oriented Organization

Advantages of this approach are:

- There could be parallel development of several projects.
- There is likely to be good communication among team members.
- This approach could enhance motivation of team members.
- The likelihood of short-term goals being realized is enhanced.
- Members of a section have a broader perspective of their assigned projects.

Disadvantages of this approach are:

- The project leader could be easily overloaded.
- The possibility of non-uniformity of standards between teams is increased. This could be mitigated by first establishing global standards for all projects.
- Members of a particular project team could be totally oblivious to other projects and by extension, the global information infrastructure. Having project rotations, where employees get the chance to work on different projects, could minimize this.

This approach is ideal for a large or medium-sized organization with a large number of projects within a given time horizon.

19.4 Hybrid (Matrix) Organization

The hybrid approach seeks to maximize the advantage of the other two approaches, while avoiding the disadvantages. This idea is to have people with default job descriptions, who can be pulled and assigned to various projects.

[Figure 19-3](#) illustrates this structure.

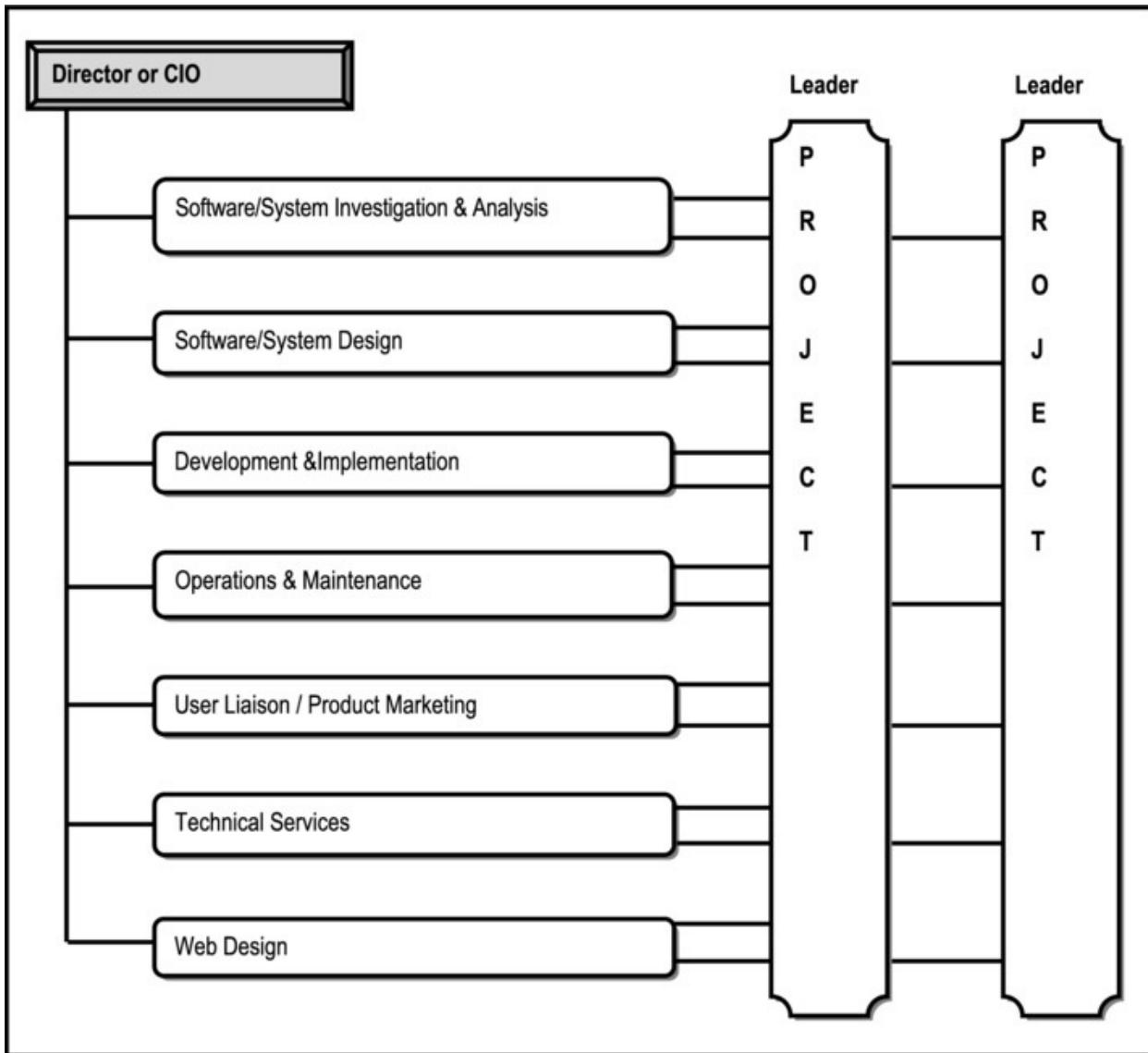


Figure 19-3. Hybrid Approach

This approach provides the following advantages:

- It facilitates assignment of teams depending on the need.
- It promotes lateral and vertical communication.
- It can be both long term and short term.

The following disadvantages may result from the approach:

- The approach could result in conflict of project priority among team members.

- There could also be conflict on reporting authority.
- Conflict on resolving project progress versus personal performance could arise.

This approach is ideal for the very large, competitive, sophisticated organization, where project teams are dynamically formed. It is ideal for companies that will specialize in several large projects within a limited time horizon. It is not as widely used as the functional approach or the project-oriented approach.

19.5 Organization of Software Engineering Firms

Software engineering firms are in the business of software construction and marketing. As you can now attest, software engineering is a very wide field with opportunities in various areas of focus. Typically, a software engineering firm focuses its attention on a finite number of product lines that are consistent with its areas of focus. This products-set will expand as the company grows and widens its scope of interest. [Figure 19-4](#) provides a summary of the software product lines for three of the leading software engineering firms in the industry. For more information on these companies, visit their respective websites as provided in section 19.8.

IBM Software Product Lines:

- Operating Systems {AIX, System i, z/OS, etc.}
- Application Servers {WebSphere Suite}
- Desktop Enterprises {CATIA, ENOVIA, SMARTTEAM, Book Manager Suite, Lotus Suite, Office Vision, etc.}
- Business Integration
- Data& Information Management {DB2 Suite}
- Instant & Team Collaboration
- Learning Software {Lotus Learning, Lotus, Virtual Classroom, Lotus Workplace, etc.}
- Security
- Networking
- Software Development {WebSphere Suite, Rational Suite, VisualAge, Suite, Tivoli Software, Lotus Software Suite, etc.}
- Wireless Technology

Microsoft Software Product Lines:

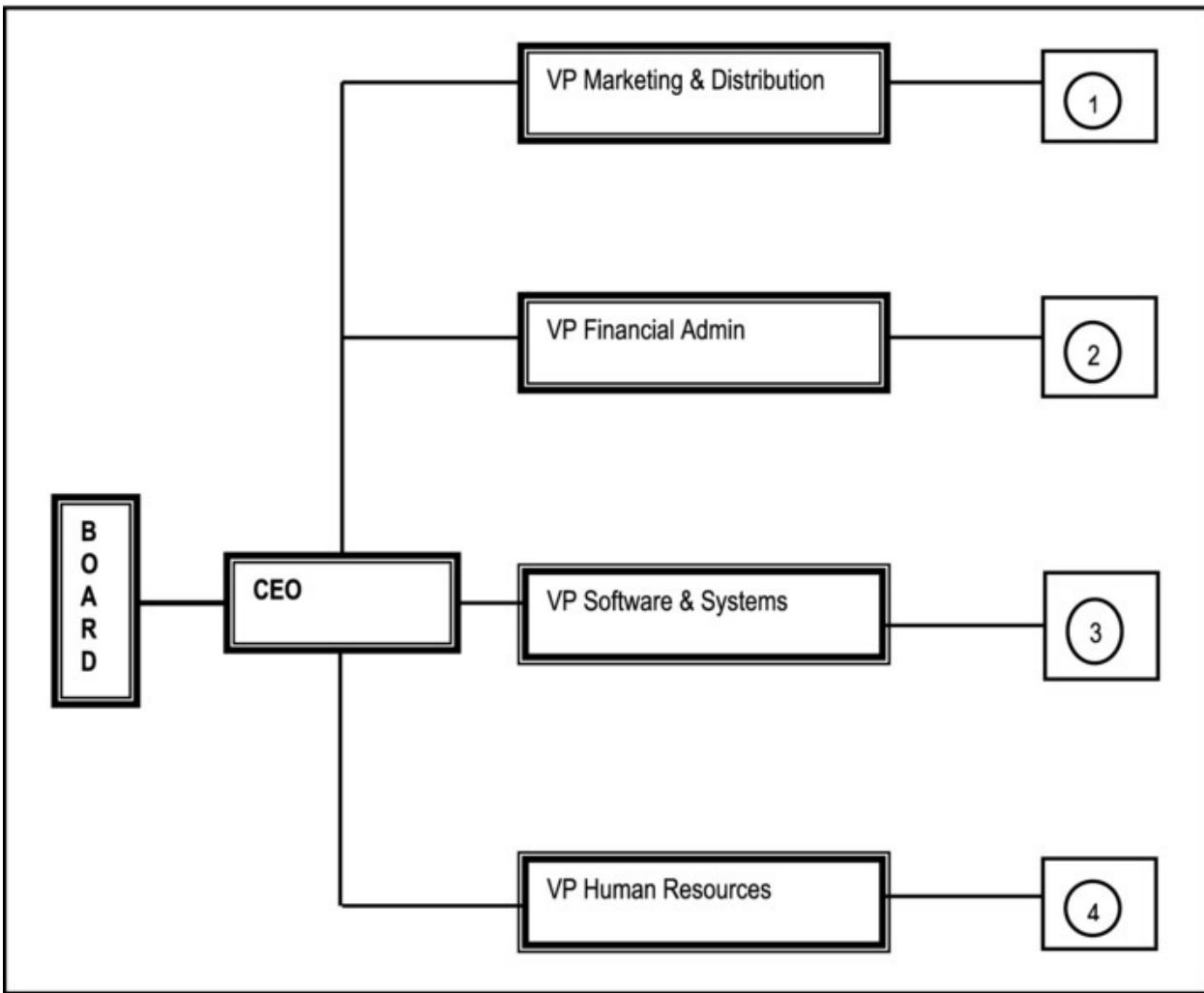
- Operating Systems {Windows Products}
- Database & Information Management {SQL server}
- Office Suite
- Developer Tools {Visual C++, C#, Visual Basic, Visual Studio, ASP.NET, etc.}
- Business Solutions
- Games & Xbox
- MSN & Other Web technologies
- Data Management {MS FoxPro, MS Access, MS SQL Server}

Oracle Software Product Lines:

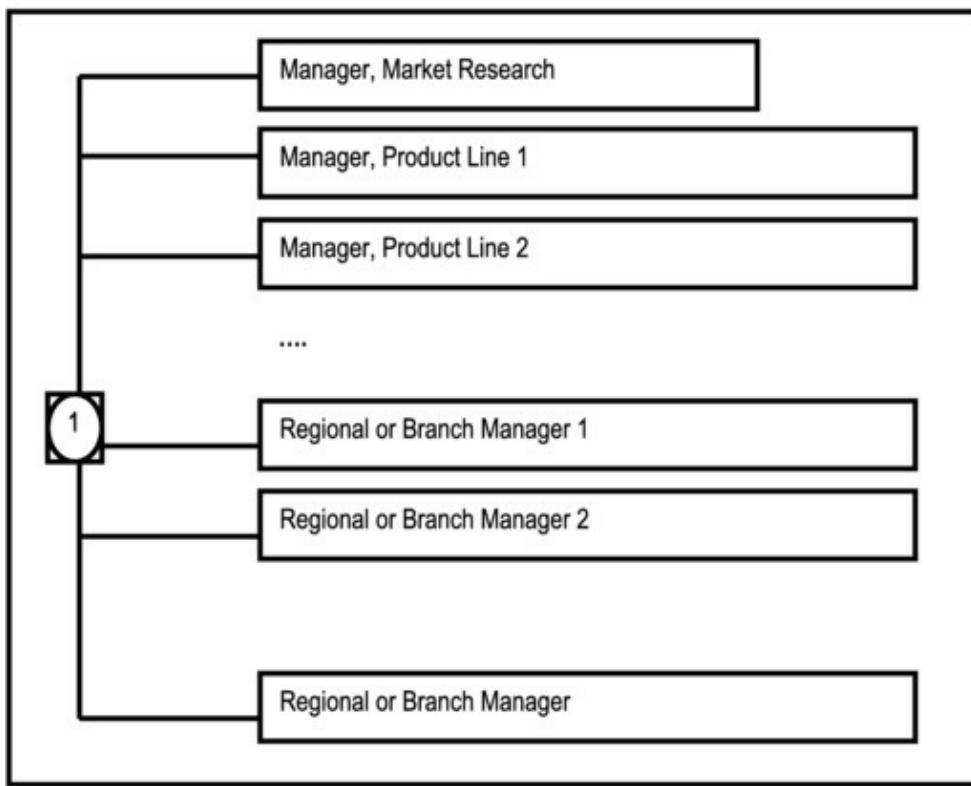
- Database & Information Management
- Applications
- Application Servers
- Development Tools
- Enterprise Manager
- Grid Computing
- Operating Systems {aided by the recent acquisition of Sun Microsystems Software}
- Application & Integration
- Application Development

Figure 19-4. Software Product Lines of Leading Software Engineering Firms

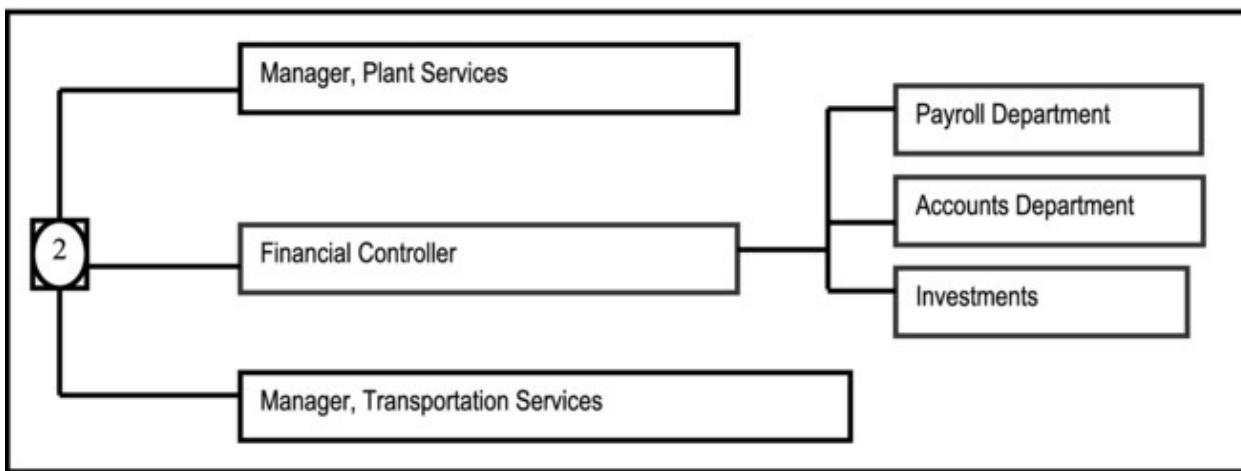
Due to the wide range of interests that these leading companies have, software engineers are necessarily placed into various teams as required. For these companies, the parallel organization and the hybrid organizations (particularly the latter) are therefore the more suited organizational approaches. Note also that in addition to writing software as one of its primary functions, these companies also need IT/IS/SE divisions/departments for their internal operations. [Figure 19-5](#) provides an illustrative organization chart that could serve a small or medium sized software engineering firm. In the figure, the Software & Systems Division could use the structure shown as the base from which individuals are drawn for various projects.



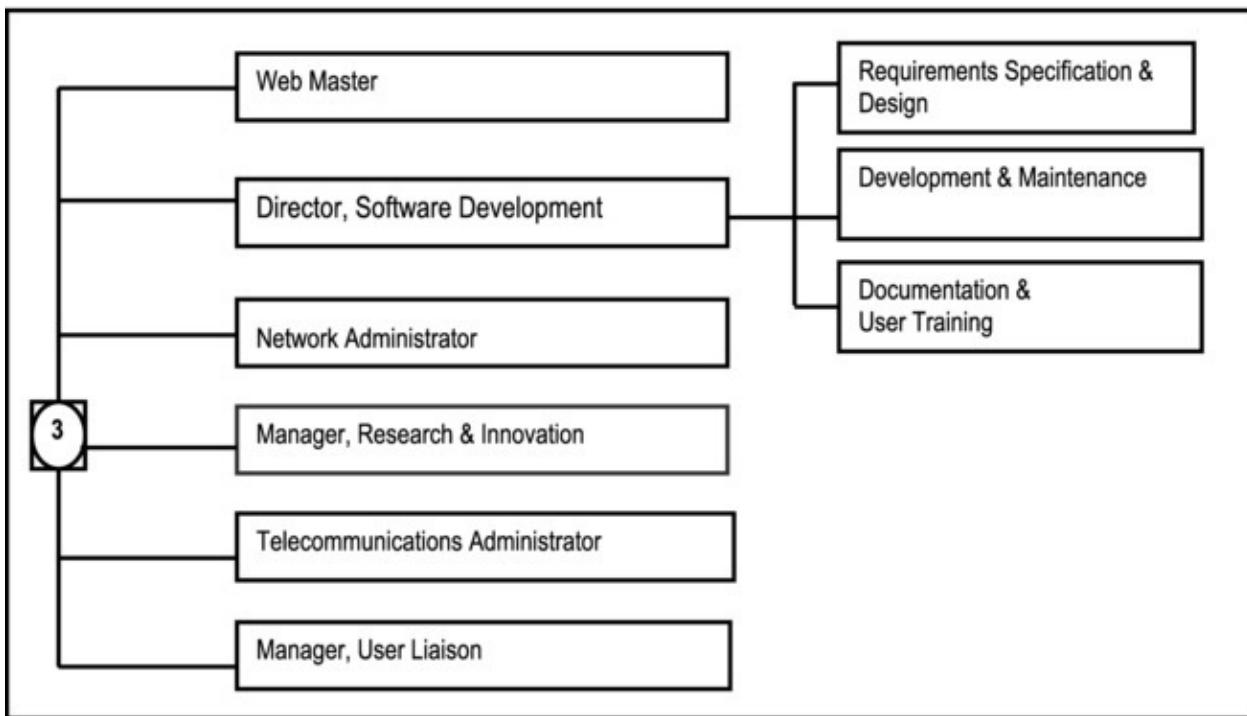
Marketing & Distribution:



Financial Administration:



Software & Systems:



Human Resource Management:

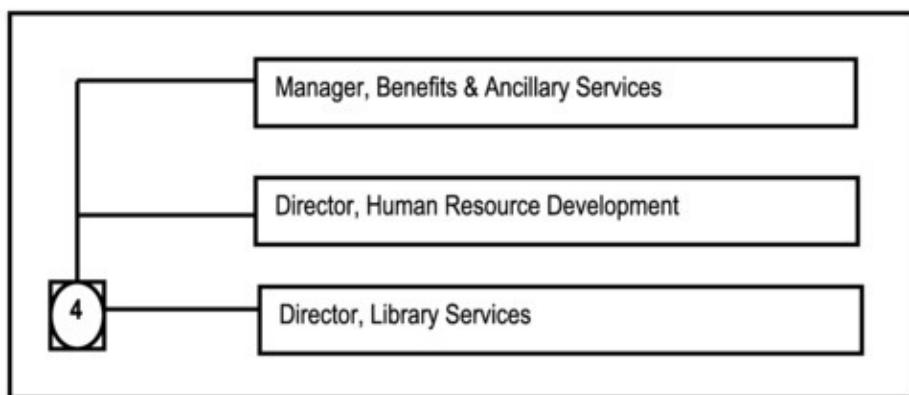


Figure 19-5. Sample Organization Chart for a Software Engineering Firm

19.6 Summary and Concluding Remarks

Let us summarize what has been covered in this chapter:

- An IT, SE, or IS division/department may be organized using the functional approach, the project-oriented approach, or the hybrid approach.
- The functional approach is ideal for a medium/small organization with a limited (manageable) number of projects in a given time horizon. However, it is widely used in small, medium, and large organizations due to its stability.
- The parallel approach is ideal for a large or medium sized organization with a large number of projects within a given time horizon.
- The hybrid approach is ideal for the very large, competitive, sophisticated organization as project teams are dynamically formed. It is ideal for companies that will specialize in several large projects within a limited time horizon. It is not as widely used as the functional approach or the project-oriented approach.
- Large software engineering companies tend to organize themselves based on product lines and geographic locations.
- Small software engineering firms tend to be functional in their organization.

This completes the course. This text has provided a methodical approach to software engineering, which if employed, will set you firmly on a path of designing and developing software systems of a high quality. Feel free to review previous chapters as needed, or to check the appendices for additional explanations and illustrations. The next chapter provides you with some sample examination questions and exercises. It has been a pleasure being your tour guide.

19.7 Review Questions

1. Describe the functional approach to organizing an IT/IS/SE Division/Department. Identify the advantages and disadvantages of the approach. State a scenario that would warrant the use of this approach.

2. Describe the project-oriented approach to organizing an IT/IS/SE Division/Department. Identify the advantages and disadvantages of the approach. State a scenario that would warrant the use of this approach.
3. Describe the hybrid approach to organizing an IT/IS/SE Division/Department. Identify the advantages and disadvantages of the approach. State a scenario that would warrant the use of this approach.
4. Which organizational structure is more suited for a medium sized or large software engineering firm? Defend your answer.

19.8 References and/or Recommended Readings

- [IBM, 2010] IBM. <http://www.ibm.com> (accessed October 2010).
- [Microsoft, 2010]. Microsoft. <http://www.Microsoft.com> (accessed October 2010).
- [Oracle, 2010]. Oracle. <http://www.oracle.com> (accessed October 2010).
- [Sprague, 1993] Sprague, Ralph H. and Barbara C. McNurlin. *Information Systems Management in Practice*. 3rd ed Eaglewood Cliffs, NJ: Prentice Hall, 1993. See [chapter 1](#).

PART F



Final Preparations

This short division has two objectives:

- To help you to review and assimilate the concepts and principles covered in the course so that you will be able to apply them to real situations
- To help you prepare for final examinations

The division consists of a final chapter that includes sample exercises and examination questions for your use.

CHAPTER 20



Sample Exercises and Examination Questions

This final chapter consists of the following:

- Introduction
- Sample Assignment 1A
- Sample Assignment 2
- Sample Assignment 3A
- Sample Assignment 4A
- Sample Assignment 5A
- Sample Assignment 6A
- Sample Assignment 7A
- Sample Assignment 8A
- Sample Interim Examination 1A
- Sample Interim Examination 2B
- Sample Final Examination A
- Sample Final Examination B

20.1 Introduction

This chapter provides you with some sample examination questions and

exercises, designed to help you solidify in your mind, the concepts and principles covered in the course. The problems are arranged in assignments and examinations. The intent is to facilitate teachers of the course, who may be using the text, as to assignments that may be given to students (of course, the teacher may make his/her own adjustments). The intent is also to enlighten students using the text, as to the type of problems they are likely to be asked to solve.

Generally speaking, assignment questions are more demanding than examination questions, and require more time. You will also observe that most of the examination questions are drawn from actual assignments. You should therefore do the assignments before attempting the examinations. The suggested weight (in points) for each question is indicated immediately following the question in curly brackets.

No solution is provided for the problems posed in this chapter, for the following reasons:

- The problems are intended to test your understanding of the materials covered. If you find that you are struggling with the solution to a problem, then you need to review the relevant sections before continuing.
- In some cases, there may be more than one solutions to the problem(s) posed. This is typical in software engineering, where you analyze alternate solutions to a problem, and choose the most prudent one.
- If you are using this book as a prescribed text for a course in software engineering, your professor will want to have a say in what questions you ought to focus on.

20.2 Assignment 1A

Question 1

- 1a. Develop an organization chart for your college/university, or any other organization with which you are familiar. {20}
- 1b. By examining the chart, propose a set of component information systems that could part of an integrated information system for the organization. {06}

Question 2

- 2a. Develop a job description for software engineer in a software engineering firm. {20}
- 2b. Propose an organization chart for the firm. {16}

Question 3

What category (or categories) of software would be best suited for the following scenarios? {13}

- 3a. An organization's financial management {03}
- 3b. Diagnosing certain respiratory and viral diseases {01}
- 3c. Aiding the development of other software {03}
- 3d. Preparation of technical documents {02}
- 3e. Assisting a senior business executive to make the right decisions {02}
- 3f. Transmitting data over a network {01}
- 3g. Building a company database {01}

Question 4

Suppose that you were on a project team to develop an integrated information system for your college or alma mater.

- 4a. Propose coding systems for the following:
 - Student Identification {02}
 - Employee Identification {02}
 - Department Code {01}
 - Course Code {02}
 - Academic Program Code {01}
- 4b. Also propose your revision of the institution's student application form. {10}

Question 5

- 5a. Give four examples of software development tools. Give four

- advantages of these products. {08}
- ba. Discuss three system life cycle models, citing the advantages, disadvantages and the situation(s) that would warrant each approach. {18}

20.3 Assignment 2

Question 1

Pay-back is defined as the time required for recovering the expenditure on a project, assuming that interest rate remains constant.

$$PB = (\text{Initial Investment}) / (\text{Cash Flow Per Annum})$$

- 1a. Suppose that a feasibility study reveals that your institution can be fully computerized in 18 months at a cost of \$4M. It is further estimated that once fully automated, the college will save \$50,000.00 per month. How long will it take to recover the initial monetary outlay, assuming that interest rates are kept constant? {04}
- 1b. Suppose that the interest rate over the next fifteen years is projected at 10% per annum. By using the payback period as a guide, conduct a NPV analysis to determine whether the project would be economically viable. What would you advise? What would be your recommendation? {08}

Question 2

Zealot Industries is a manufacturing and distribution firm that currently faces severe competition. It is fast losing once loyal customers to its competitors. At a management retreat, it was determined that the main reason for the company's fate is due to Zealot's outdated information infrastructure. Further, the managers agreed that the company needed to acquire a comprehensive integrated system that spans all critical aspects of the firm's operation. The problem is, due to Zealot's uniqueness (which the company wants to preserve), there is no readily available software product that would meet the company's needs. The following

alternatives were identified:

- A. Purchase a product for \$560,000.00, and customize it. Estimated customization cost unclear; estimated implementation time not less than one year.
- B. Hire a software engineering firm for the entire project for \$1.2 M. Estimated implementation time 18 months, but modules will be available (on a phased basis) after the first 9 months.
- C. In-house development via Information Systems Division (ISD). Estimated technology cost of \$700,000.00 to be added to the cost of staffing the ISD. Estimated implementation time similar to alternative B.

As Zealot's lead software engineer on the project, construct an evaluation grid that will facilitate objective and thorough analysis of each alternative and determination of the most prudent one. Explain how you would proceed with the analysis and determination. {16}

■ **Hint** You have two alternate approaches:

Alternative 1: Remember your quality factors and your feasibility factors (use two evaluation grids or a grid that merges both sets of factors).

Alternative 2: Remember your PDM, TELOS and MURRE factors (use one evaluation grid).

Question 3

- 3a. What are the methodologies for obtaining information that are available to the software engineer? {06}
- 3b. A survey is required to determine the symptoms of certain categories of diseases: sexually transmitted diseases (STD's), viral diseases, respiratory diseases. Twelve doctors are to be consulted.

Would you use interview or questionnaire? Defend your choice. {02}

- 3c. Develop a questionnaire or interview question list (depending on your answer to the previous question) to capture the information required. {12}

Question 4

The manager of an auto-shop wants to be at least 90% certain that there is a bug in the current stock management module of his inventory system, before considering to replace it. He contracts an analyst to advise him. After sampling thirty (30) items of the same unit of measure, the analyst runs each through the system and obtains a closing stock. He summarizes his findings as follows:

Sample Size 30
Margin of error ... +/- 1 unit
Standard deviation ... 2.795
Mean closing stock ... 10 units
Should the manager replace the suspicious module? Show your reasoning.
{04}

■ **Note** 95% => $Z = 1.96$ 99% => $Z = 2.58$

Question 5

- 5a. What is the first deliverable of a software engineering project?
State and briefly clarify its contents. {20}
- 5b. What is the second deliverable of a software engineering project?
State and briefly clarify its contents. {24}
- 5c. Describe a methodology for evaluating and selecting project alternatives. {12}
- 5d. What are the essential issues to consider in a feasibility study?
{12}

Question 6

Suppose that you were the lead software engineer on a project to design and develop a software system to facilitate on-line registration of students of your college/university.

- 6a. By conducting a set of brain-storming sessions or otherwise, construct a preliminary list of object types (information entities) that you would expect to be represented in this system. For each entity, provide a brief description of the type of data that it will host, and the operations to be defined on it. {40}
- 6b. Propose an information topology chart that represents your on-line

- registration system. {12}
- 6c. Propose a user interface topology chart that represents your on-line registration system. {20}
- 6d. One important object type that must be in your model is the **Student** object type. Propose a finite state machine for **Student**, showing the different states that a **Student** instance may be in. {08}

20.4 Assignment 3A

Question 1

- 1a. When would you use a HIPO chart as opposed to a process oriented flowchart (POF)? {04}
- 1b. State the main use of the procedure analysis chart. {02}
- 1c. Compare the process oriented flow chart with the DFD, identifying the main difference(s) and similarity(ies). {06}
- 1d. A variety store manager wishes to computerize its operation. The main issues to address are as follows:
 - Purchases from external vendors, signaled by invoices.
 - Purchases affect the purchase log, as well as accounts payable and the inventory logs.
 - Inventory management as affected by purchases and sales.
 - Every inventory item belongs to a category.
 - Sale of goods to customers (credit sales) as well as cash sales. Sales affect the sales log, as well as the inventory log.

From the information given, develop a POF or a DFD of a proposed system. {10}

Question 2

ABC Inc. is a software engineering firm offering accredited training in

Information technology (in various specific areas) to the public. The company is currently computerizing its training program. The following table shows some information and processes to be managed.

Process	Input Form/Entity	Output Entity/Report
Process Application	Application Form	Applicant Detail Entities + Edit Listing
Add Company Courses	Company Course Input Form	Courses Master + Edit Listing
Update / Delete Company Courses	Company Course Update / Delete Request Form	Courses Master + Edit Listing
Enter Course Schedule	Company Course Schedule Input Form	Course Schedule + Edit Listing
Modify Course Schedule	Company Course Schedule Modification Form	Course Schedule + Edit Listing
Accept Course Appraisal	Course Appraisal Form	Course Appraisal + Edit Listing
Accept Student Performance	Student Performance Form	Student Performance + Edit Listing

Figure 20-1. ABC Process Plan

- 2a. From the information given, develop a POF for the system (the chart should indicate what the "Application Detail Files" might be). {12}
- 2b. Identify some possible outputs that may be obtained from the system and include them on the chart. {08}

Question 3

BDF inventory management includes the following key operations:

- Purchase orders are sent to company suppliers whenever raw materials are needed. These are logged in the purchase order file.
- Purchases from suppliers, signaled by purchase invoices.
- Purchases (made from company suppliers) affect the purchase invoice log (file), as well as Accounts payable (file) and the inventory file.
- Inventory management as affected by purchases and sales.
- Every inventory item belongs to a category.
- Large customers periodically send sale orders in,

requesting certain furniture pieces. These are logged in the sale order file.

- Sale of goods to customers (credit sales) as well as cash sales. Sales affect the sales invoice log (file), the accounts receivable file, and the inventory file.
- 3a. From the information given, identify all inputs and information entities in the system. {13}
 - 3b. Use the information to develop a POF or a DFD of the Inventory Management System. {14}
 - 3c. Propose an information topology chart for the system. {10}

Question 4

Suppose that the **Student** entity is part of a college information system. A **Student** instance may be in a state of applied, accepted, oriented, registered, on-probation, on-leave, or graduated. Additionally, in given state, the instance may be modified. Propose an FSM for the **Student** entity. {16}

20.5 Assignment 4A

Question 1

- 1a. Clearly outline the steps involved in conducting a PERT-CPM analysis. {08}
- 1b. Explain how a PERT diagram and a Gantt chart may be used to assist in the management of a software engineering project. {04}

Question 2

A project involving the installation of a computer system consists of eight activities as shown in the activity table below:

Activity	Immediate Predecessor	Time (weeks)
A Equipment acquisition & Preparation	--	3
B Data conversion & Testing	--	6
C User Training	A	2
D Parallel Run	B, C	5
E Troubleshooting & Observation	D	4
F Fine-tuning	E	3
G Advanced User Training	B, C	9
H Signoff	F, G	3

Figure 20-2. Sample Project A

- 2a. Draw the PERT network for this project. {14}
- 2b. Identify the critical path. {02}
- 2c. What is the expected completion time for the project? {02}
- 2d. Identify three activities that can be delayed and for each, determine the slack. {03}

Question 3

LMX Software considering developing a new software system. The project activities identified so far are shown below:

Activity	Immediate Predecessor	Time (weeks)
A Prepare Requirements Specification	--	6
B Prepare Design Spec	--	8
C Prepare Development Team	A, B	12
D Develop & Test Module 1	C	4
E Develop & Test Module 2	C	6
F Develop & Test Module 3	D, E	15
G Conduct Comprehensive System Test	E	12
H Refine & Hand Over System	F, G	8

Figure 20-3. Sample Project B

- 3a. Develop a PERT network for the project. {14}
- 3b. Identify the critical path. {02}
- 3c. Determine the total project duration. {02}

3d. Identify three activities that can be delayed and for each, determine the slack. {03}

20.6 Assignment 5A

Question 1

- 1a. Compare OOD to FOD showing the major points of divergence. {04}
- 1b. Give four advantages of OOD over FOD. {04}
- 1c. Describe in your own words, four approaches to architectural design and cite a situation that would warrant each approach. {16}

Question 2

- 2a. Why are software standards important (give six reasons)? {06}
- 2b. State ten issues to be addressed in software standards. {10}

Question 3

- 3a. State six major problems that are likely to occur from poor database design. {06}
- 3b. The following is an excerpt from the requirements for a college academic administration system:
 - Courses are offered by various departments without any overlap (a department offers anywhere between 5 and 30 courses).
 - The courses make up academic programs, in some instances a course may occur in more than one program. Academic programs are offered by departments (no overlap allowed).
 - A faculty typically consists of several departments.
 - A lecturer is scheduled to lecture at least two courses. Each course is lectured in a specific lecture room.

- A student may register for several courses; typically a course is pursued by a minimum of fifteen students.
- Each student is registered to one department only.

From the information given, develop an ERD. {14}

Question 4

- Explain how you would proceed to categorize operations to be developed for a software system. {05}
- How important is operations specification? If you were in charge of operations specification on a software engineering project, what methodology would you use, and what detail would you specify for each operation? {15}

Question 5

The **Student** entity (object type) of the question 3 contains the following attributes: **IdNbr**, **Name**, **Gender**, **DateOfBirth**, **Major** and **DeptNbr**. Each student has a unique identification number (**IdNbr**). The department (**DeptNbr**) to which a student is assigned must previously exist in the **Department** entity. Also, the student's major must reside in the **Academic Program** entity. **Gender** must be male or female and **DateOfBirth** must be a valid date in the 20th or 21st century.

Propose an EOS for the operation to allow addition of valid student records (your outline may include either a Warnier-Orr diagram or pseudo code). {20}

20.7 Assignment 6A

Question 1

The Inventory Management Information System (IMIS) of a marketing company has the following database specification:

- The company has several warehouses, each storing certain inventory items without overlap.
- The company has a cadre of suppliers, each supplying various items of inventory, with possible overlap.

- The company purchases items by first sending a purchase order to a supplier (of course, the supplier could receive several orders). Each purchase order details the items required. In responding to the purchase order, the supplier submits an invoice, detailing the items supplied, along with other relevant information.
- The company may also sell items from its inventory. In such a case, a sale-invoice is submitted to the customer, which details the items sold, along with other relevant information.
- A sale-invoice is usually with respect to a sale-order, received from a customer. A sale-order is essentially a purchase order, coming into the company, from one of its customers.
- Each inventory item belongs to a category.
- A department may make a requisition for inventory items. In response, inventory items may be issued to department(s).

- 1a. From this information, develop an ERD. {30}
- 1b. By conducting a brainstorming session (or otherwise), and using your E-R diagram as well as guidelines in [Figure 10-8 of Chapter 10](#), construct an initial Entity Specification Grid (ESG) for the IMIS project. {100}
- 1c. Propose an EOS for an operation to inquire on inventory items in the system. {20}

Question 2

- 2a. Briefly describe the three alternatives to structuring the help system of contemporary software. {06}
- 2b. Briefly describe two approaches to managing messages that a software product might send to the end-user. {04}
- 2c. Briefly describe the six stages in the design of real-time systems. {12}
3. During the requirements specification of an integrated MIS for a hotel, the following information was documented:

System Name: RBL Hotel Administrative Management Information System

Main System Modules:

1. Supplies Subsystem involves the management of:
 - Supplier Business Information
 - Rooms Log
 - Merchandise Items
2. Customer Subsystem involves the management of:
 - Customer Business Information
 - Customer Reservations Log
 - Customer Actual Service Log
 - Customer Cancellations Log
3. Financial Management Subsystem involves the management of:
 - Purchase Orders
 - Purchases
 - Purchase Returns
 - Payments Made
 - Sales
 - Payments Received
 - Investments

Figure 20-4. Hotel Management System Overview

- 3a. Assuming that the information given describe object types (or entities) in each sub-division, and that details are readily available, show how you would proceed to design a menu driven user interface, using principles of OOD (do not attempt to add any new entity). {14}
- 3b. What enhancement(s) would you make to your basic design in order to accommodate experienced users as well as novices? {04}

20.8 Assignment 7A

Question 1

- 1a. Outline a procedure for maintaining excellent software quality. {05}
- 1b. State and briefly clarify the software quality factors. {12}
- 1c. Apart from software quality, state three other issues that are important during the development of a software system. {03}

Question 2

- 2a. Why is human resource management important in a software engineering project? {04}
- 2b. From a software engineering perspective, choose your four favorite management styles. For each, briefly describe the approach, then state the advantages and disadvantages. {24}

Question 3

- 3a. If you were the lead software engineer for a software engineering firm, and you had the opportunity of hiring programmers and junior software engineers, what precautions would you follow to ensure the selection of the best candidates? {05}
- 3b. State and briefly clarify five criteria that you would use to assess the candidates. {05}

Question 4

- 4a. Summarize three imperatives that a software engineer should pursue in creating a desirable working environment. {06}
- 4b. Summarize the steps involved in creating and preserving accountability during a software engineering project. {04}
- 4c. “Succession planning is naive and foolish. Why should you train someone else to relieve you of your job?” Discuss this moot. {04}

20.9 Assignment 8A

Question 1

- 1a. What is the difference between software cost and software price? {02}
- 1b. What factors affect software engineering cost? {04}
- 1c. What factors affect software price? {06}

Question 2

- 2a. What factors affect software value? {03}
- 2b. Summarize three approaches for assessing software productivity. {12}
- 2c. Provide an overview of one popular software estimation model. {10}

Question 3

- 3a. State and briefly clarify six critical issues to be addressed during the implementation of a software system. {12}
- 3b. Suppose that you were in charge of implementing a comprehensive software system for your institution or alma mater. Your new system actually replaces several splinter systems that were used; these systems were independent and not integrated. Your new system is integrated, involving various aspects of the college operation.

By carefully assessing each implementation factor (as stated in the previous question), explain how you would proceed with the implementation. {18}

Question 4

- 4a. State and briefly clarify six factors that are likely to affect the cost of maintaining a software system. {06}
- 4b. State four reasons legacy systems are prevalent in the business community. Also describe three approaches for treating such systems. {07}
- 4c. State and briefly clarify five activities that are likely to take place during software re-engineering. {05}

- 4d. Describe a scenario that would warrant the use of software integration techniques. {02}

20.10 Sample Interim Examination 1A

Answer all four questions

Question 1

- 1a. What is software engineering and how does it relate to the management of an organization? {03}
- 1b. Propose a job description for a software engineer in a software engineering firm. {14}
- 1c. Name three of the key organizational divisions you would expect in such a firm. {03}

Question 2

- 2a. Suppose that you were on a project team to develop an integrated information system for your institution. Propose coding systems for the following:
 - Student Identification {02}
 - Employee Identification {02}
 - Department Code {02}
 - Course Code {02}
- 2b. A Software Engineering firm is offering to the public, accredited training in IT. It is desirable to design an application form for would be trainees. The information provided on the form is to be keyed into the computer and stored in the company's database. Using established principles of forms design, sketch what your application form might look like.

Your application form should capture the following information:

name, address, telephone number, name and address of next of kin, educational record and qualification, course preference(s), marital status, sex, dexterity, date of birth, height, weight, Employment history, current job, career ambition, references (three), and criminal record. {12}

Question 3

- 3a. What is the purpose of a feasibility study? Briefly elaborate on the essential issues to be considered in a feasibility study. {13}
- 3b. How would you proceed to evaluate system alternatives and recommend the most prudent alternative? {04}
- 3c. State three essential elements to be considered when evaluating system cost? {03}

Question 4

- 4a. “Problem definition is not critical to the success of a software engineering venture; what is required is smart programming.” Discuss this moot. {05}
- 4b. What is the first deliverable in a software engineering project? Briefly discuss the main components of this deliverable. {15}

20.11 Sample Interim Examination 2B

Answer all four questions.

Question 1

- 1a. State four (4) alternate approaches to information gathering that are available to the software engineer, citing a situation that would warrant each approach. {04}
- 1b. A survey is required to determine the symptoms of certain categories of diseases — sexually transmitted diseases (STD's), viral diseases, respiratory diseases. Twelve doctors are to be

consulted.

State four (4) alternate approaches to information gathering that are available to the software engineer, citing a situation that would warrant each approach. {04}

- 1c. Develop a questionnaire or interview question list (depending on your answer to the previous question) to capture the information required. {14}

Question 2

BDF inventory management includes the following key operations:

- Purchase orders are sent to company suppliers whenever raw materials are needed. These are logged in the purchase order file.
 - Purchases from suppliers, signaled by purchase invoices.
 - Purchases (made from company suppliers) affect the Purchase Invoice Log (file), as well as accounts payable (file) and the inventory file.
 - Inventory management as affected by purchases and sales.
 - Every inventory item belongs to a category.
 - Large customers periodically send sale orders in, requesting certain furniture pieces. These are logged in the sale order file.
 - Sale of goods to customers (credit sales) as well as cash sales. Sales affect the Sales Invoice Log (file), the Accounts Receivable file, and the Inventory file.
- 2a. From the information given, identify all inputs and information entities in the system. {13}
 - 2b. Use the information to develop a POF or a DFD of the Inventory Management System. {14}
 - 2c. Propose an information topology chart for the system. {10}

Question 3

3a. In structured decision systems, what occasion(s) would warrant the use of the following?

- Structured English {02}
- Decision Table {02}
- Decision Tree {02}

3b. When would you use an extended-entry decision table as opposed to a simple decision table? {04}

3c. Reduce the following decision table to the minimum number of rules. {10}

Conditions / Actions	Rules															
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Sufficient Qty on Hand?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Qty Large Enough for Discount?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Wholesale Customer?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Sales Tax Exemption Files?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Ship Items & Prepare Invoice	X	X	X	X	X	X	X	X								
Set up Back Order									X	X	X	X	X	X	X	X
Deduct Discount	X	X														
Add Sales Tax		X	X	X		X	X	X								

Figure 20-5. Decision Table for Sale of Products

Question 4

LMX Software considering developing a new software system. The project activities identified so far are shown below:

Activity	Immediate Predecessor	Time (weeks)
A Prepare Requirements Specification	--	6
B Prepare Design Spec	--	8
C Prepare Development Team	A, B	12
D Develop & Test Module 1	C	4
E Develop & Test Module 2	C	6
F Develop & Test Module 3	D, E	15
G Conduct Comprehensive System Test	E	12
H Refine & Hand Over System	F, G	8

Figure 20-6. LMX Software Project

- 4a. Develop a PERT network for the project. {14}
- 4b. Identify the critical path. {02}
- 4c. Determine the total project duration. {02}
- 4d. Identify three activities that can be delayed and for each, determine the slack. {03}

20.12 Sample Final Examination A

This examination consists of four (4) sections. You are required to do all questions in Section A and one question from each of the other three sections.

Section A: Answer all questions from this section.

- 1.1 What is software engineering, and how does it relate to management of an organization? {03}
 - 1.2 State four (4) categories of computer software to which you have been exposed. {04}
 - 1.3 Examine a software life cycle model: state the basic concept, identify two advantages, two disadvantages, and a situation that would warrant the use of such a model. {07}
 - 1.4 If you were in charge of building a comprehensive information system for your institution, which life cycle model would you use and why? {02}
 - 1.5 Suppose that you were on a project team to develop an integrated software system for your institution. Propose coding systems for the following: {04}
 - Employee Identification
 - Course Code
- 2.1 State the first, and three other deliverables of a software project. Mention four important ingredients of the first deliverable. {08}
 - 2.2 Describe one method of evaluating system alternatives. Be sure to

mention the evaluation criteria used in your methodology. {12}

Section B: Answer one question from this section.

- 3.1 When would you use a HIPO chart as opposed to a POF? {04}
- 3.2 Compare the process-oriented flow chart with the DFD, identifying three similarities and three differences. {06}
- 3.3 A variety store manager wishes to computerize its operation. The main issues to address are as follows:
 - Purchases from external vendors, signaled by invoices.
 - Purchases affect the Purchase Log (file), the Accounts Payable (AP) file, and the Inventory file.
 - Inventory Management as affected by purchases and sales.
 - Every inventory item belongs to a category.
 - Sale of goods to customers affects the Sales Log file, the Accounts Receivable file, and the Inventory file.

From the information given, develop POF or a DFD of a proposed system. {10}

- 4.1 In structured decision systems, what occasion(s) would warrant the use of the following?
 - Structured English {02}
 - Decision Table {02}
 - Decision Tree {02}
- 4.2 When would you use an extended-entry decision table as opposed to a simple decision table? {04}
- 4.3 Reduce the following decision table to the minimum number of rules. {10}

Conditions / Actions	Rules															
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Sufficient Qty on Hand?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Qty Large Enough for Discount?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Wholesale Customer?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Sales Tax Exemption Files?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Ship Items & Prepare Invoice	X	X	X	X	X	X	X	X								
Set up Back Order									X	X	X	X	X	X	X	X
Deduct Discount	X	X														
Add Sales Tax		X	X	X		X	X	X								

Figure 20-7. Decision Table for Sale of Products

Section C: Answer one question from this section.

5.0 LMX Software considering developing a new software system.
The project activities identified so far are shown below:

Activity	Immediate Predecessor	Time (weeks)
A Prepare Requirements Specification	--	6
B Prepare Design Spec	--	8
C Prepare Development Team	A, B	12
D Develop & Test Module 1	C	4
E Develop & Test Module 2	C	6
F Develop & Test Module 3	D, E	15
G Conduct Comprehensive System Test	E	12
H Refine & Hand Over System	F, G	8

Figure 20-8. LMX Software Project

- 5.1 Develop a PERT network for the project. {14}
- 5.2 Identify the critical path. {02}
- 5.3 Determine the total project duration. {01}
- 5.4 Identify two activities that can be delayed and for each, determine the slack. {03}
- 6.0 The following is an excerpt from the requirements for a college Academic Administration System:

- Courses are offered by various departments, with the understanding that a course can be offered by only one department (a department typically offers between 5 and 30 courses).
- The courses make up academic programs, in some instances a course may occur in more than one program. Academic programs are offered by departments with the understanding that a program can be offered by only one department.
- A school typically consists of several academic departments.
- A lecturer is scheduled to lecture at least two courses. Each course is lectured in a specific lecture room.
- A student may register for several courses; typically a course is pursued by a minimum of fifteen students.
- Each student is registered to one department only.

Figure 20-9. Sample Database Requirements for Academic Administration System

- 6.1 From this information, develop an ERD. {14}
- 6.2 State two products that could be used to implement the database for this system. {02}
- 6.3 State four problems that are likely to occur if the database is not correctly designed. {04}

Section D: Answer one question from this section.

- 7.1 Contrast object-oriented design (OOD) with function-oriented design (FOD), showing the main areas of divergence. {04}
- 7.2 Give four advantages of OOD over FOD. {04}
- 7.3 During the requirements specification of an integrated MIS for a hotel, the following information was documented:

System Name: RBL Hotel Administrative Management Information System

Main Subsystems:

1. Supplies Subsystem involves the management of:
 - Supplier Business Information
 - Supplier Locations
 - Rooms Log
 - Merchandise Items
2. Customer Subsystem involves the management of:
 - Customer Business Information
 - Customer Reservations Log
 - Customer Actual Service Log
 - Customer Cancellations Log
3. Financial Management Subsystem involves the management of:
 - Purchase Orders
 - Purchases
 - Purchase Returns
 - Payments Made
 - Sales
 - Payments Received
 - Investments

Figure 20-10. Hotel Management System Overview

Assuming that the information given describe object types (or entities) in each sub-division, and that details are readily available, show how you would proceed to design a menu driven user interface, using principles of OOD (do not attempt to add any new entity). {12}

- 8.1 Give four advantages of software standards. {04}
- 8.2 State four key issues that software standards should address. {04}
- 8.3 How do standards relate to quality assurance during software development. {08}
- 8.4 Apart from standards and quality assurance, what other issues/challenges must be resolved during software development? {04}
- 9.0 LMX Software Inc. has developed comprehensive information system to take care of various aspects of administration of a

college/university. The company is preparing to implement the system at Knox College. Knox College currently relies on a system that was developed in the early 1980s; it is highly unreliable and incomprehensive.

- 9.1 What operating environment would you recommend for the system? Justify your answer. {03}
- 9.2 In LMX Software's implementation plan, who should train the senior administrators and the clerical staff respectively? Defend your answer. {03}
- 9.3 What other issues should LMX Software pay close attention to? State these issues, and for each, provide some guidelines for the LMX Software Team. {10}
- 9.4 The existing system at Knox College is an example of a legacy system. Provide two reasons such systems are prevalent. Also briefly describe two other alternatives (not mentioned in the case) for dealing with such systems. {04}

20.13 Sample Final Examination B

This examination consists of four (4) sections. You are required to do all questions in Section A and one question from each of the other three sections.

Section A: Answer all questions from this section.

- 1.1 What is software engineering? {02}
- 1.2 For each of the following scenarios, state what category (or categories) of software would be best suited. {06}

Diagnosing certain respiratory and viral disease:

Aiding the development of other software:

Preparation of technical documents:

Assisting a senior business executive to make the right decisions:

Transmitting data over a network:

Building a company database:

1.3 State six primary functions, and two critical roles of the software engineer. {08}

Functions:

- a.
- b.
- c.
- d.
- e.
- f.

Roles:

- a.
- b.

1.4 State four software life cycle models. {04}

- a.
- b.
- c.
- d.

2.1 State four important deliverables of a new software engineering project. {04}

- a.
- b.
- c.
- d.

2.2 “Problem definition is not critical to the success of a software engineering venture; what is required is smart programming.” Briefly discuss this moot. {04}

3.0 Below are two lists of terms. Each term in **list 1** describes an aspect of software engineering, and has at least two terms in **list 2**

that directly relate to it. However, not all the terms in **list 2** directly relate to terms in **list 1**. For each term in **list 1**, indicate the terms in **list2** that directly relate to it. {12}

List 1:	List 2:
Product Documentation:	System help facility Software development kits Operational environment Information topology chart CASE tools Enhancements Marketing System manuals Categorization of operations Training Corrective changes Users' guide Decision tables Object types RAD tools
Software Modifications:	
Software Implementation Issues:	
Diagramming Techniques:	
Software Development Tools:	

Figure 20-11. Software Engineering Puzzle

Section B: Answer one question from this section.

- 4.1 What is the main objective of a feasibility study? {01}
- 4.2 State three methods of information gathering that are available to the software engineer. {03}
- 4.3 Zealot Industries is a manufacturing and distribution firm that currently faces severe competition. It is fast losing once loyal customers to its competitors. At a senior management retreat, it was determined that the main reason for the company's fate is due to Zealot's outdated information infrastructure. Further, the managers agreed that the company needs to acquire a comprehensive integrated software system that spans all critical aspects of the firm's operation. The problem is, due to Zealot's uniqueness (which the company wants to preserve), there is no

readily available software product that would meet the company's needs. The following alternatives were identified:

- A. Purchase a product for \$560,000 and customize it. Estimated customization cost unclear; estimated implementation time not less than one year.
- B. Hire a software engineering firm for the entire project for \$1.2 M. Estimated implementation time 18 months, but modules will be available (on a phased basis) after the first 9 months.
- C. In-house development via Information Systems Division (ISD). Estimated technology cost of \$700,000.00 to be added to the cost of staffing the ISD. Estimated implementation time similar to alternative B.

As Zealot's lead software engineer on the project, construct an evaluation grid that will facilitate objective and thorough analysis of each alternative and determination of the most prudent one. Explain how you would proceed with the analysis and determination.
{16}

Hint You have two alternate approaches:

Alternative 1: Remember your quality factors and your cost factors (use two evaluation grids).

Alternative 2: Remember your PDM, TELOS and MURRE factors (use one evaluation grid).

5.0 ABC Inc. is a software engineering firm offering accredited training in information technology (in various specific areas) to the public. The company is currently computerizing its training program, developing a *Training Information Management System* (TIMS). The figure below shows some user functions to be managed. Use the information provided in the figure to answer the following questions:

Process	Input	Output
Process Application	Application Form	Applicant Detail Files + Edit Listing
Add Company Courses	Company Course Input Form	Courses Master File + Edit Listing
Update/Delete Company Courses	Company Course Update/Delete Request Form	Courses Master File + Edit Listing
Enter Course Schedule	Company Course Schedule Input Form	Course Schedule File + Edit Listing
Modify Course Schedule	Company Course Schedule Modification Form	Course Schedule File + Edit Listing
Accept Course Appraisal	Course Appraisal Form	Course Appraisal File + Edit Listing
Accept Student Performance	Student Performance Form	Student Performance File + Edit Listing

Figure 20-12. Summary of Some Operations of the TIMS, ABC Inc.

- 5.1 Develop a process POF for the system (the chart should indicate what the “Application Detail Files” might be). {12}
- 5.2 Include on the chart, some possible outputs that may be obtained from the system. {08}

Section C: Answer one question from this section.

One central data entity (or object type) in any computerized Human Resource Management System (HRMS) is the Employee entity. All other entities revolve around it. For instance:

- An employee may have several dependents.
- Each employee is assigned a classification, a department, and a job description. Of course, a classification, department, or job description may apply to several employees.
- Each employee has an employee history, an academic history and professional qualifications.
- Each employee has a work log, as well as a payment log.

- 6.1 From the information given, develop an ERD for the HRMS. {14}
- 6.2 Propose a Warnier-Orr diagram or pseudo-code for adding

employees to the company database. To do this, you may assume that your algorithm will include a sub-routine for validating non-key attributes of the employee record (you do not need to show this sub-routine). Also assume that there are sub-routines (which you do not need to write) for adding other records (in respective entities) which are related to the employee record. {06}

- 7.0 A project involving the installation of a computer system consists of eight activities as shown in the activity table below:

Activity	Immediate Predecessor	Time (weeks)
A Equipment acquisition & Preparation	--	3
B Data conversion & Testing	--	6
C User Training	A	2
D Parallel Run	B, C	5
E Troubleshooting & Observation	D	4
F Fine-tuning	E	3
G Advanced User Training	B, C	9
H Signoff	F, G	3

Figure 20-13. Final Examination B Sample Project

- 7.1 Draw the PERT network for this project. {14}
- 7.2 Identify the critical path. {02}
- 7.3 What is the expected completion time for the project? {02}
- 7.4 Identify three activities that can be delayed and for each, determine the slack. {03}

Section D: Answer one question from this section.

- 8.1 Briefly describe two alternatives to structuring the help system of contemporary software. {04}
- 8.2 Briefly describe two approaches to managing messages that a software product might send to the end users. {04}
- 8.3 Briefly describe the six stages in the design of real-time systems. {12}
- 9.1 Give four advantages of software standards. {04}

- 9.2 State four key issues that software standards should address. {04}
- 9.3 How do standards relate to quality assurance during software development? {08}
- 9.4 Apart from standards and quality assurance, what other issues/challenges must be resolved during software development? {04}
- 10.0 Suppose that Amigo Software Inc. has been contracted to computerize the operation of your institution, and you are the lead software engineer on the team:
- 10.1 Describe the operating environment you would prescribe for the institution. Should it be distributed or centralized? Defend your answer. {04}
- 10.2 Your software system is to replace two legacy systems called Quo-Data and White-Horse respectively. Do you anticipate conducting code conversions? Defend your answer. How would you proceed? {04}
- 10.3 What strategy would you use for changing from the old systems to the new system? Defend your answer and explain how you would proceed. {04}
- 10.4 Who would you assign the responsibility of training the department heads, the secretaries, faculty members (who are not department heads), and student workers respectively? {04}
- 10.5 Identify four (4) aspects of the system that you would strongly recommend for subsequent Web accessibility. {04}

PART G



Appendices

This final division contains additional information that the more curious reader may peruse. It includes additional topics in OOM, excerpts from the initial system requirements, the requirements specification, and the design specification for an inventory management system. The chapters are:

- Appendix 1 — Introduction to Object-Oriented Methodologies
- Appendix 2 — Basic Concepts of Object-Oriented Methodologies
- Appendix 3 — Object-Oriented Information Engineering
- Appendix 4 — Basic Guidelines of Object-Oriented Methodologies
- Appendix 5 — Categorizing Objects
- Appendix 6 — Specifying Object Behavior
- Appendix 7 — Tools for Object-Oriented Methodologies
- Appendix 8 — Excerpts from the ISR of the Inventory Management System
- Appendix 9 — Excerpts from the RS of the Inventory Management System
- Appendix 10 — Excerpts from the DS of the Inventory Management System



APPENDIX 1



Introduction to Object Oriented Methodologies

This chapter contains:

- Software Revolution and Rationale for Object-Oriented Techniques
- Information Engineering and the Object-Oriented Approach
- Integrating Hi-tech Technologies
- Characteristics of Object-Oriented Methodologies
- Benefits of Object-Oriented Methodologies
- Summary and Concluding Remarks

A1.1 Software Revolution and Rationale for Object-Oriented Techniques

One serious anomaly in information technology is the advancement of hardware technology disproportionately to software technology. We are in the sixth generation of computer hardware; we are in the fourth generation (perhaps the fifth depending on one's perspective) of computer software.

Software generations include the following:

- Machine code
- Assembly language
- High-level languages; databases based on hierarchical and network approaches
- Relational systems, 4GL's, CASE tools, and applications
- I-CASE, object-oriented techniques, multi-agent applications, and intelligent systems

One urgent concern in the software industry today is to create more complex software, faster and at lower costs. The industry demands at reduced cost, a quantum leap in:

- Complexity
- Reliability
- Design capability
- Flexibility
- Speed of development
- Ease of change
- Ease of usage

How can the software engineering industry respond to this huge demand? How can software engineers be equipped to produce software of a much higher quality than ever demanded before, at a fraction of the time? A few years ago, companies would be willing to invest millions of dollars into the development of business applications that took three to five years to be fully operational. Today, these companies demand immediate solutions, and they want more sophistication, more flexibility, and more convenience.

As the software engineering industry gropes for a solution to the demands of 21st century lifestyle, a principle worth remembering is, “*Great engineering is simple engineering*” (see [Martin, 1993]). Object-oriented (OO) techniques simplify the design of complex systems; so far, it is the best promise of the quantum leaps demanded by industry. Through OO techniques, complex software can be designed and constructed block-by-block, component-by-component. Moreover, tested and proven components can be reused multiple

times for the construction of other systems, potentially spiraling into a maze of complex, sophisticated software which was once inconceivable to many.

OO methodologies (OOM) involve the use of OO techniques to build computer software. It involves *OO analysis and design* (OOAD), as well as *OO programming languages* (OOPLs). It also involves the application of *object-oriented technology* (OOT or OT) to problem-solving.

A1.2 Information Engineering and the Object-Oriented Approach

Traditionally, software systems have been developed through a life cycle model approach of investigation, analysis, design, development, implementation and maintenance. These activities were separate, often done by different people. Further, systems were developed in a piecemeal fashion — roughly on a department-by-department basis. This traditional approach still prevails in a number of organizations.

Information engineering (IE) is about looking at the entire enterprise as a system of interrelated sub-systems, then proposing and engineering a strategic information system that meets the corporate goals and objectives of the enterprise. It is more dynamic, challenging and rewarding than the traditional approach. As an illustration, the CUAIS (college/university administrative information system) information topology chart (ITC) of [Figure A1-1](#), provides excerpts from a comprehensive, integrated information system for a college/university, as opposed to a department-wise automation.

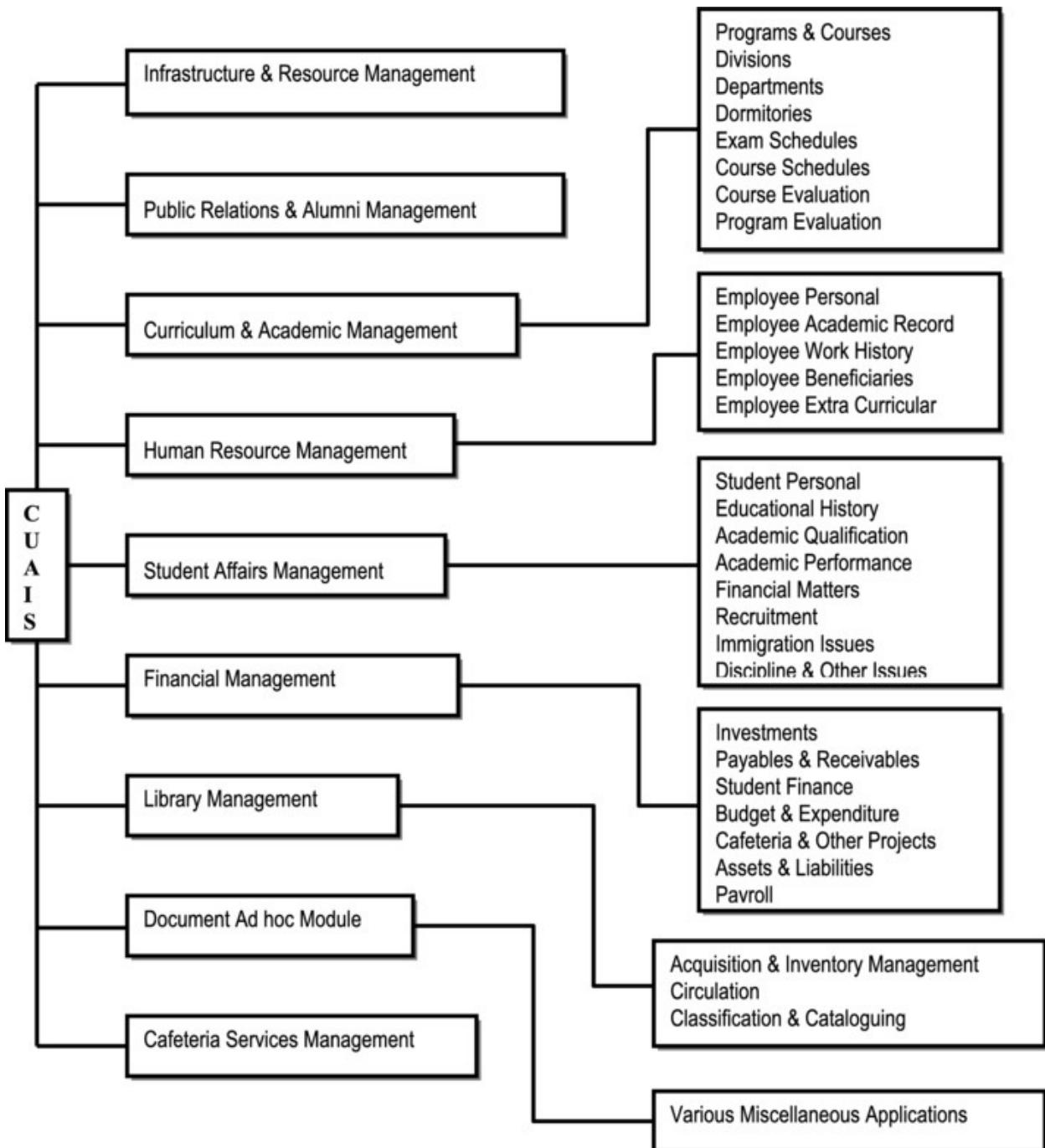


Figure A1-1. Information Topology Chart for the CUAIS Project

Whereas the traditional approach is usually managed at the middle management level, enterprise-wide engineering must be managed at the executive level, if it is to be successful. The information technology (IT) professional in charge must be very experienced, well versed in information systems, and able to inspire confidence at all levels of the organization.

We can apply OOM to information engineering in a way that enables us to

~~We can apply OOM to information engineering in a way that enables users~~
to conceptualize an information system as a set of interacting, inter-related component objects. At the highest level, the entire system is a large object. The component objects encompasses encapsulated states (data), methods and means of communicating with each other as well as with other external objects, all this being transparent to the end-user.

When OOM is applied to information engineering, the process becomes more effective and rewarding. We use the term OO information engineering (OOIE) to describe this scenario. In OOIE, the modeling process is more integrated, involving the previously disjointed activities of investigation, design, development, implementation and maintenance — assuming the use of OO-ICASE tools (which will be discussed in [Appendix 7](#)).

Another term that you will hear bandied around is *OO enterprise modeling* (OOEM). Some authors like to parse and dissect and come up with differences between OOIE and OOEM. This course makes no such distinction. OOIE and OOEM essentially describe the same thing. Use of one term over another is a matter of personal preference. [Appendix 3](#) will revisit the topic and provide more details.

A1.3 Integrating Hi-tech Technologies

OO analysis and design (OOAD) is not OO programming (in the early days of OOM, the two were mistaken to be the same). Any high-level language can implement an OO design, some more elegantly than others. An OO high level language (which will be subsequently discussed) facilitates easy implementation of certain of OO concepts, for example: class, inheritance, encapsulation, etc. So do OO-CASE tools.

OOAD facilitates the integration of a number of leading technologies (so called hi-tech) towards the generation of complex systems, with a number of advantages as spin-offs. The technologies include:

- CASE and I-CASE
- Visual Programming via OO programming languages (OOPLs)

- Code Generation
- Repository and Class Libraries
- Repository-Based Methodologies
- Information Engineering
- OO Databases
- Non-procedural languages
- Formal (Mathematically-Based) Methods
- Client-Server Technology

According to Martin’s prediction, “the sooner OOAD becomes widespread, the sooner software creation will progress from a cottage industry to an engineering discipline” (see [Martin, 1993]). We are in the midst of that transformation.

A1.4 Characteristics of Object-Oriented Methodologies

[Figure A1-2](#) provides some fundamental characteristics of OOM: These characteristics lead to various benefits as mentioned in the upcoming section.

- OO methodologies change the way we conceptualize software systems: We view a system as a set of interacting objects (components) rather than a set of interacting functions. This is more natural than traditional function-oriented approach; it is easier to think about objects and their behavior than functions and their procedures.
- OO methodologies naturally support CBSE: Systems can often be constructed from existing system components that have already been tested and proven. Hence, there is a high degree of reusability, shortened development time, and increased reliability.
- Through OOM, more complex systems can be constructed with less effort.
- OO techniques fit naturally with CASE technology.
- OO techniques promote the use of a CASE repository — an ever-growing library of object types. These object types are designed to be customizable to different system needs.
- OO techniques lead to easier and more likely creation of systems that work correctly. As the saying goes, OO techniques are “more modular than modular programming design.”

[*Figure A1-2. Characteristics of OOM Summarized*](#)

A1.5 Benefits of Object-Oriented Methodologies

Object technology offers many benefits to the software engineering industry and the business environment. The more prominent ones are mentioned in [Figure A1-3](#). Indeed, OO methodologies have become commonplace in the software engineering industry. To attempt software engineering in a manner that is oblivious to these methodologies would be at best extremely risky.

- **Reusability of Code:** A tested system component can be reused in the design of another component.
- **Stability and Reliability:** Software can be constructed from tested components. Organizations can be assured of guaranteed performance of software.
- **More Sophistication:** Through powerful OO-CASE tools and CBSE, more complex systems can be constructed.
- **Understandability:** Designer and user think in terms of object and behavior rather than low-level functional details. This results in more realistic modeling that is easier to learn and communicate.
- **Faster Design:** Most RAD tools and contemporary CASE tools are object oriented to some degree. Also code reusability enhances faster development.
- **Higher Quality Design:** New software can be constructed by using tested and proven components.
- **Easier Maintenance:** Since systems are broken down into manageable component objects, isolation of system faults is easy.
- **Dynamic Lifecycle:** I-OO-CASE tools integrate all stages of the software development life cycle (SDLC).
- **Interoperability:** Classes may come from different vendors.
- **Platform Independence:** Classes may be designed to operate and/or communicate across different platforms.
- Better communication between IT professionals and business people.

Figure A1-3. Benefits of OOM

A1.6 Summary and Concluding Remarks

Let us summarize what we have covered so far:

- There is a disproportional advancement in computer hardware when compared to computer software. Nonetheless, the software engineering industry has made quantum leaps and has achieved an astounding lot over the past five decades.
- To write quality software in the twenty first century is a very

challenging experience: Users are demanding more complexity, flexibility and functionality for less cost and within a much shorter timeframe. Because of this, software engineering via traditional methods is an early non-starter. OOM and OOSE present the opportunity to meet the demands of the current era.

- Information engineering (IE) is the act of engineering the information infrastructure of an organization via OO methodologies. The result is a more efficient organization and information infrastructure.
- OOM facilitates the integration of various technologies for the more efficient operation of the organization.
- OOM brings a number of significant advantages to the software engineering industry and the business environment.

Against this background, we may now proceed to a discussion of the fundamentals of the OO approach to software engineering. This will be done in the next chapter.

A1.7 Recommended Readings

[Due, 2002] Due', Richard T. *Mentoring Object Technology Projects*. Saddle River, New Jersey: Prentice Hall, 2002. See [chapters 1 and 2](#).

[Lethbridge, 2005] Lethbridge, Timothy C. and Robert Laganiere. *Object-Oriented Software Engineering 2nd ed.* Berkshire, England: McGraw-Hill, 2005. See [chapter 1](#).

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1993. See [chapters 1 and 3](#).

APPENDIX 2



Basic Concepts of Object-Oriented Methodologies

This chapter provides clarification on the fundamental concepts in OOM. Since they form the basis of OO software, it is imperative that the student has a clear understanding of them. The chapter includes the following sections:

- Objects and Object Types
- Operations
- Methods
- Encapsulation and Classes
- Inheritance and Amalgamation
- Requests
- Polymorphism and Reusability
- Interfaces
- Late Binding
- Multithreading
- Perception versus Reality
- Overview of the Object-Oriented Software Engineering Process
- Summary and Concluding Remarks

A 2.1 Objects and Object Types

22.1 Objects and Object types

An *object* is a concept or thing about which data can be stored, and upon which a set of operations is applicable. The concept of an object is similar to that of an entity in the relational model. Here, however, the definition includes possible actions (operations) which may be performed on the object. In the relational model, this is not the case.

Object may be tangible or intangible (conceptual). Examples of tangible objects are:

- A book
- A building
- A chair
- A motor vehicle engine
- A student
- An employee

Examples of abstract (conceptual) objects are:

- A shape in a program for drawing
- A screen with which the user interacts
- An engineering drawing
- An airline reservation
- An employee's work history

An object may be composed of other objects, which in turn may be composed of other objects and so on.

Example 1:

- A car consists of a chassis, body, engine, and electrical system. The engine consists of pistons, engine block, input manifold, etc.
- A computer system is made up of various components including processor(s), data buses, control buses, magnetic storage units, I/O devices, memory, etc.
- A software system may be composed of subsystems, each consisting of various objects and facilities.

An *object type* refers to a group or category of (related) objects. The instances of an object type are the actual objects. An object type differs from an entity (in relational model) in three ways:

- The object type involves encapsulation of data structure and operation.
- This means the use of potentially more complex data structures.
- For an object type, there is an emphasis on inheritance.

Example 2:

If **Employee** is an object type, then the person Bruce Jones could be an instance (object) belonging to object type **Employee**.

It is often the case that the term “object” is used loosely to mean either an instance of an object type, or the object type itself. In such cases, the context should be used to determine what the applicable definition is.

A2.2 Operations

Operations are the means of accessing and manipulating the data (instances) of an object (type). Since the operations are defined on the object type, they necessarily apply to each instance of that object type. If for instance, we define the operations **Hire**, **Modify**, **Remove**, **Retire**, **Fire**, **Resign**, **Inquire**, **Print** on an object type, **Employee**, then for each employee object, these operations are applicable.

Operations of an object type (should) reference only the data structure of that object type. Strict OO design forbids an operation to directly access the data structure any object type other than its own object type. To use the data structure of another object type, the operation must send a *request* (message) to that object.

In a truly OO software (CASE or programming language) environment, operations and data of an object type are encapsulated in a class (more on this later). Several quasi OO software development environments exist however, that typically facilitate the creation of a relational database, with an OO GUI sitting

on top (we will revisit this concept in [appendix 7](#)).

In the absence of an OO software development tool, operations may be implemented as programs (with singular functions), some of which may manipulate a database. Obviously, this is a less desirable situation.

In the spirit of the course (review [chapter 12](#)), we may expand the meaning of an operation as described above, to mean a set of related activities. If this is done, then an operation may be implemented as a class consisting of related sub-operations.

A2.3 Methods

Methods specify the way in which operations are encoded in the OO software. Typically, they are implemented as functions and/or procedures, depending on the development software. The ability to create and maintain methods related to classes is one significant advantage of OO-CASE tools (more on this later).

In OO software, methods are stored within classes, each method having its parent class. The methods are accessible to object instances of that class. *Static methods* are accessible to instances of the host class as well as other classes. In non-OO software, methods translate to procedures and/or functions of application programs.

In environments such as Java, an *interface* is defined as a collection of service declarations with no implementation. Services that are generic (i.e. applicable to objects of various object types) may be implemented as interfaces. Implementation details are left for the specific objects which utilize such interfaces. We will revisit interfaces later on.

 **Note** Some OOPs and hybrid languages use other terms for methods. For instance, C++ uses *functions* and Pascal uses *procedures*. For the duration of this course, we will continue to use the term method.

Example 3:

Consider the class **Employee**, encompassing several object types:

- **Employee-Personal-Data**
- **Academic-Log**
- **Extracurricular**
- **Employment-History**

The operation, **Create-Employee-Application** would require a number of sub-operations such as:

- **Create-Employee-Personal-Data**
- **Create-Applicant-Academic-Log**
- **Create-Applicant-Extracurricular**
- **Create-Applicant-Employment-History**

Each of these component sub-operations would have other constituent methods associated with them. For instance, **Create-Employee-Personal-Data** must allow for data entry and validation, as well as write a record (an object instance) to the data structure, **Employee-Personal-Data**.

Notice how a complex operation is naturally broken down into simple manageable activities. This simple illustration, underscores the observation that OO techniques naturally lead to more organized software systems than do the more traditional techniques.

A2.4 Encapsulation and Classes

What do we mean by *encapsulation*, and what is a *class*? These two concepts are closely related to each other, so that it is difficult to discuss one without reference to the other.

A2.4.1 Encapsulation

Encapsulation is the packaging together of data and operations. It is the result (act) of hiding implementation details of an object from its user. The object type (class) hides its data from other object types and allows data to be accessed only via its defined operations — this feature is referred to as *information hiding*.

Encapsulation separates object behavior from object implementation. Object implementations may be modified without any effect on the applications using them.

A2.4.2 Class

A *class* is the software implementation of an object type. It has data structure

and methods that specify the operations applicable to that data structure.

In an OO software environment, object types are implemented as classes. In a procedural programming environment, object types are implemented as modules.

Example 4:

In the CUAIS model, we may have an **Employee** class of an **Employment Subsystem** with options to **Hire**, **Modify**, **Fire**, **Remove**, **Resign**, **Retire**, **Inquire**, or **List** employees. These would be defined as operations on the **Employee** class.

Note that even in the absence of an OO software environment, OOSE affects the user interface topology of the system. More significantly, it affects the fundamental design of the software being constructed.

A2.5 Inheritance and Amalgamation

Inheritance and amalgamation are two critical principles of OO methodologies. They both affect code reusability, and by extension, the productivity of the software design and construction experiences. This section briefly examines each principle.

A2.5.1 Inheritance

A class may inherit properties (attributes and operations) of a *parent class*. This is called *inheritance*. The parent class is called the *super-class* or *base class*; the inheriting class is called the *sub-class* or *child class*.

A class may qualify as both a sub-class and a super-class at the same time. Additionally, a class may inherit properties from more than one super-class. This is called *multiple inheritances*.

Example 5:

SoftwareEngineer may be designed to be a sub-class of the super-class **Faculty-Member**, which is in turn a sub-class of **Employee**, thus making **Faculty-Member** a sub-class and super-class at the same time. **Employee** may be designed to inherit from **CollegeMember**, thus making it a sub-class and super-class.

Student may be designed to inherit from **CollegeMember**. **StudentWorker** may be designed to inherit from both **Student** and **Employee**.

As you will later see, multiple inheritances pose potential problems to software implementation. For this reason, it is not supported in some software development environments. It may therefore be prudent to avoid multiple inheritances, depending on the software limitations.

A2.5.2 Amalgamation

An object may be composed of other objects. In OOM, we refer to such an object as an *aggregate* or *composite* object. The act of incorporating other component objects into an object is called *aggregation* or *composition*. Since this is done through the object's class, the class is also called an aggregation (or composition) class. Throughout this course, we shall use the term amalgamation to mean an aggregation and/or composition.

Example 6:

Following on from Example 5, **Employee** may be designed to be composition of **EmployeePersonalInfo**, **EmployeeEmploymentHistory**, **EmployeeCompensation** and **EmployeeAcademicLog**.

A2.6 Requests

To make an object do something, we send a *request* to that object. The request is conveyed via a message. The message contains the following:

- Object name (of target object)
- Operation
- Optional parameters
- The type of the value the request (service call) returns

In an OO environment, objects communicate with each other by making requests and sending them via messages. The software is designed to respond to messages. Objects respond to requests by returning values and evoking certain operations intrinsic to such objects.

A2.7 Polymorphism and Reusability

An operation or object may appear in different forms; this is referred to as *polymorphism*. Operational polymorphism may occur in any of the following ways:

- Overriding an inherited operation
- Extending an inherited operation
- Restricting an inherited operation
- Overloading an operation

Overriding: A sub-class may override the features of an inherited operation to make it unique for that sub-class.

Extending: A sub-class may *extend* an inherited operation by adding additional features.

Restricting: A sub-class may restrict an inherited operation by inhibiting some of the features of the inherited operation.

Overloading: A class may contain several versions of a given operation. Each version (implemented as a method) will have the same name but different parameters and code. Depending on the argument(s) supplied when the operation is invoked, the appropriate version will run.

Example 7:

Suppose that the subclass **Student** inherits from super-class **College Member**. **College Member** defines a **PRINT** operation which is inherited by **Student**. **Student**'s **PRINT** operation may be extended to include printing attributes of **Student** which are not attributes of **College Member**.

Polymorphism presumes inheritance; polymorphic objects and/or methods can be defined only within the context of an inheritance hierarchy. However, note that the converse does not necessarily hold: inheritance does not necessarily mean that polymorphism is in play.

The most powerful spinoff from inheritance and polymorphism is reusability of code. Reusability may occur in any of the following ways:

- A method (or data structure) is inherited and used by a subclass
 - A method is inherited and extended in a sub class
 - A method is inherited and restricted in a sub class
 - A method is copied and modified in another method
-

Note Reusability and polymorphism are not miracles; they must be planned. Reusability of classes may span several systems, not just one. To this end, class libraries are of paramount importance. The OO-CASE tool should therefore support class libraries.

A2.8 Interfaces

An interface is a class-like structure with the following exceptions:

- The data items of an interface are constants only.
- The interface contains method signatures only.

The idea of an interface is to promote polymorphism and facilitate a safe alternative to multiple inheritances. Polymorphism is achieved when different classes *implement* (a term used for interfaces instead of inherit) the interface and override its methods (rather, method signatures). Multiple inheritances occur when a particular class implements several interfaces, or inherits a super-class and implements one or more interfaces.

A2.9 Late Binding

Late binding is the ability to determine the specific receiver (requester) and the corresponding method to service the request at run time. In traditional programming, a procedure or function call is essentially translated to a branch to

a particular location in memory where that section of instructions executes. As part of the compilation process, the necessary pieces of the program are put (bound) together before program execution. This process is referred to as (early) *binding*.

In the object oriented paradigm, no concern is given to which object will request any given service of another object. Further, objects (more precisely classes) are not designed to anticipate what instances of other classes will require their services. There therefore needs to be late binding between a message (request) and the method that will be used to respond to that request.

A2.10 Multithreading

The principle of *multithreading* has been immortalized by the Java programming language. However, software industry leaders have latched on to the concept, so much so that we now talk about multithreading operating systems, as well as multithreading software applications.

Multithreading is the ability of an object to have concurrent execution paths. The effect is to speed up processing and thereby enhance throughput and hence, productivity. Multithreading is one of the reasons object technology offers better machine performance.

A2.11 Perception versus Reality

Something is said to be *transparent* if it appears not to exist, when in fact it does. For example, calculations, cross-linking of data, procedures for methods, etc., should be transparent to the user.

Something is said to be *virtual* if it appears to exist, when in fact, it does not. A good example of this concept is the underlying code that facilitates the OO concept. We are making the computer (software) behave like humans do, as opposed to the traditional approach of thinking like the computer.

Methods are not actually stored in each object —this would be wasteful. Instead, the software examines each request that refers to an object, and selects the appropriate methods (code) to execute. The method is part of the class (or a higher class in the hierarchy), not part of the object. Yet the software gives the illusion that they are, thus creating a virtual environment for the user.

A2.12 Overview of the Object-Oriented Software Engineering Process

Software is constructed using one of two approaches:

- The function-oriented approach
- The object-oriented approach

The function-oriented approach is the traditional approach, based on a life cycle model. The software system passes through several distinct phases, as summarized by the software development life cycle (SDLC):

- Investigation
 - Object Structure Analysis (OSA)
 - Object Behavior Analysis (OBA)
 - Class Structure Design (CSD)
 - Operations Design (OD)
 - Implementation
 - Maintenance
- } Each phase employs and builds on standard set of techniques and representations of the software system. The combined set of activities is referred to as modeling.

The object oriented approach is the contemporary approach to software construction. The software system passes through phases which are integrated by a standard set of techniques and representations, as summarized by the following revised SDLC:

- Investigation
 - Analysis
 - Design
 - Development
 - Implementation
 - Maintenance
- } Each of these phases employs different techniques and representations of the software system.

There are two significant advantages of this revised SDLC over the traditional one:

- In the traditional approach, the symbols used in representing the system requirements vary with the different phases, thus presenting the possibility for confusion. In the OO approach, the symbols used are consistent throughout the entire engineering process. The risk for confusion is therefore minimized.
- In the OO approach, there is no need for a distinction between the design phase and development phase. This is particularly true if an OO-ICASE tool is employed. The combination of analysis and design activities in the OO paradigm constitutes system modeling, which often results in the automatic generation of code (typically in an OOPL) which can then be accessed and modified.

Software construction via the OO paradigm is an integrated, pragmatic approach, which is very exciting and rewarding. It is hoped that you will catch this excitement and be part of the software revolution.

A2.13 Summary and Concluding Remarks

Here is a summary of what we have covered in this chapter:

- An object type is the term used to describe a family of like objects. The object type defines the data items and operations that can be applied to objects of that family. An object is a concept of thing about which data can be stored and upon which certain operations can be applied. Each object is an instance of the object type that it belongs to.
- An operation is a set of related activities to be applied to an object. Operations are implemented as methods or classes with related methods. A method is a set of related

instructions for carrying out an operation.

- A class is the implementation of an object type or an operation. It encapsulates the structure (i.e. data items) and methods for all instances of that class.
- In an OO environment, objects communicate with each other by making requests and sending them via messages.
- An object inherits all the properties (attributes and methods) of its class. A sub-class inherits properties from a super-class. A class may qualify as both a sub-class and a super-class at the same time. A class may also inherit from multiple super-classes.
- Polymorphism is the act of an operation or object behaving differently in different circumstances. It is implemented via method overriding, method extension, method restriction and method overloading.
- An interface is a class-like structure that consists of constants and/or method signatures only. It facilitates polymorphism and a safe alternative to multiple inheritances.
- Late binding is the ability to determine the specific receiver (requester) and the corresponding method to service the request at run time.
- Multithreading is the ability of an object to have concurrent execution paths. The effect is to speed up processing and thereby enhance throughput and hence, productivity.
- Something is transparent if it appears not to exist, when in fact it does. Something is virtual if it appears to exist when in fact, it does not.
- With the introduction of OOM, the SDLC has been simplified and more reliable software products can be constructed.

The next chapter starts the process of delving deeper into OOM. The approach employed in this course is top-down: We will start by looking at the organization as a large complex object type, then work downwards into the details.

A2.14 Recommended Readings

[Lee, 2002] Lee, Richard C. and William Tepfenhart. *Practical Object-Oriented Development With UML and Java*. Upper Saddle River, New Jersey: Prentice Hall, 2002. See [chapters 1](#) and [2](#).

[Lethbridge, 2005] Lethbridge, Timothy C. and Robert Laganiere. *Object-Oriented Software Engineering 2nd ed.* Berkshire, England: McGraw-Hill, 2005. See [chapter 2](#).

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1993. See [chapter 2](#).

APPENDIX 3



Object-Oriented Information Engineering

This chapter discusses object-oriented information engineering (OOIE) as the starting point in developing an information infrastructure for the organization. We will start by looking at the organization as a large complex object type, then work downwards into the details. The chapter includes the following sections:

- Introduction
- Engineering the Infrastructure
- Diagramming Techniques
- Enterprise Planning
- Business Area Analysis
- System Design
- System Construction
- Summary and Concluding Remarks

A3.1 Introduction

Software engineering as we know refers to the discipline of specifying, designing, constructing and managing computer software. Information engineering is heavily influenced by software engineering but takes a different perspective: it refers to a set of interrelated disciplines required to build a computerized enterprise based on information systems, and managing the

resulting information infrastructure. *Information engineering* (IE) concerns itself with the planning, construction and management of the entire information infrastructure of the organization. It typically pulls several software engineering projects and other related projects together, and provides general direction to the organization. You might be familiar with the term *business process re-engineering* (BPR). Information engineering (also referred to as enterprise engineering) includes that concept and much more. Another term that is widely used is *OO enterprise modeling* (OOEM). As mentioned in [appendix 1](#), for practical reasons, we may consider OOIE and OOEM to be identical, so the rest of the discussion will stick with OOIE.

The following are some broad objectives of IE:

- Support of the management needs of the organization
- Aligning the information infrastructure to the corporate goals of the organization
- Increasing the value and credibility of the computerized systems
- Effective management of the information so that it is available whenever required
- Increasing the speed and ease of software application development
- Reducing the rigors of software systems maintenance
- Achievement of integration of component systems
- Facilitation and promotion of reusable design and code
- Promotion of a higher level of communication and understanding between IS professionals and non-IS professionals in the organization

Desirable characteristics of IE include the following:

- It involves higher user participation.
- It involves the use of sophisticated software planning and development tools (CASE tools, DBMS suites and RAD tools).
- It embraces CBSE through integration of component

systems.

- It leads to an organization that is more progressive, efficient and technologically aware.

A3.2 Engineering the Infrastructure

In planning an organization's information infrastructure, the following three important ingredients must be considered:

- The structure of the system's required object types
- The behavior of the required system operations
- The underlying technology required to support the organization's mission and operations

James Martin's information system (IS) pyramid (see [Martin, 1989a] and [Martin, 1993]) proposes four levels of planning that address these ingredients:

- Enterprise Planning
- Business Area Analysis
- System Design
- System Construction

According to Martin, these four strands of information management must relate to the three sides of the pyramid — relating to structure, operations, and technology. These concepts are summarized in [Figure A3-1](#). Upcoming sections of this chapter will discuss each strand of information management, but first, a discussion of diagramming techniques is in order.

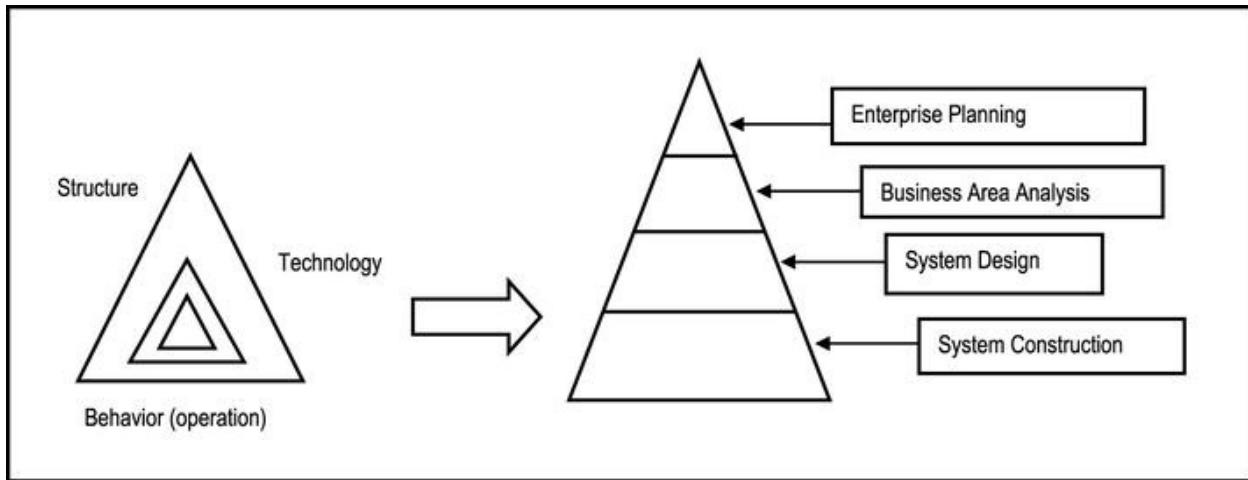


Figure A3-1. Overview of Martin's IS Pyramid

A3.3 Diagramming Techniques

If you are going to make any significant progress in designing the information infrastructure, you will first need to put together a number of technical documents. Covering large volumes of information in a succinct, comprehensive manner is critical; hence the importance of diagrams.

Figure A3-2 provides a list of recommended diagramming techniques that are applicable to OOSE. Most of these techniques have been discussed or mentioned in the course. In the interest of additional clarity, some of them will be revisited, and the new ones discussed in the upcoming sections of this and subsequent chapters of the appendices. There are other diagramming techniques that are used in OOSE, but these are the ones that this course recommends. At the highest level of planning the information infrastructure of a business enterprise, or architecture of a software system, object flow diagrams and information topology charts are useful. From this point, you then proceed down into the structure of the various object types that comprise the system. Then you address the behavior of instances of each object type.

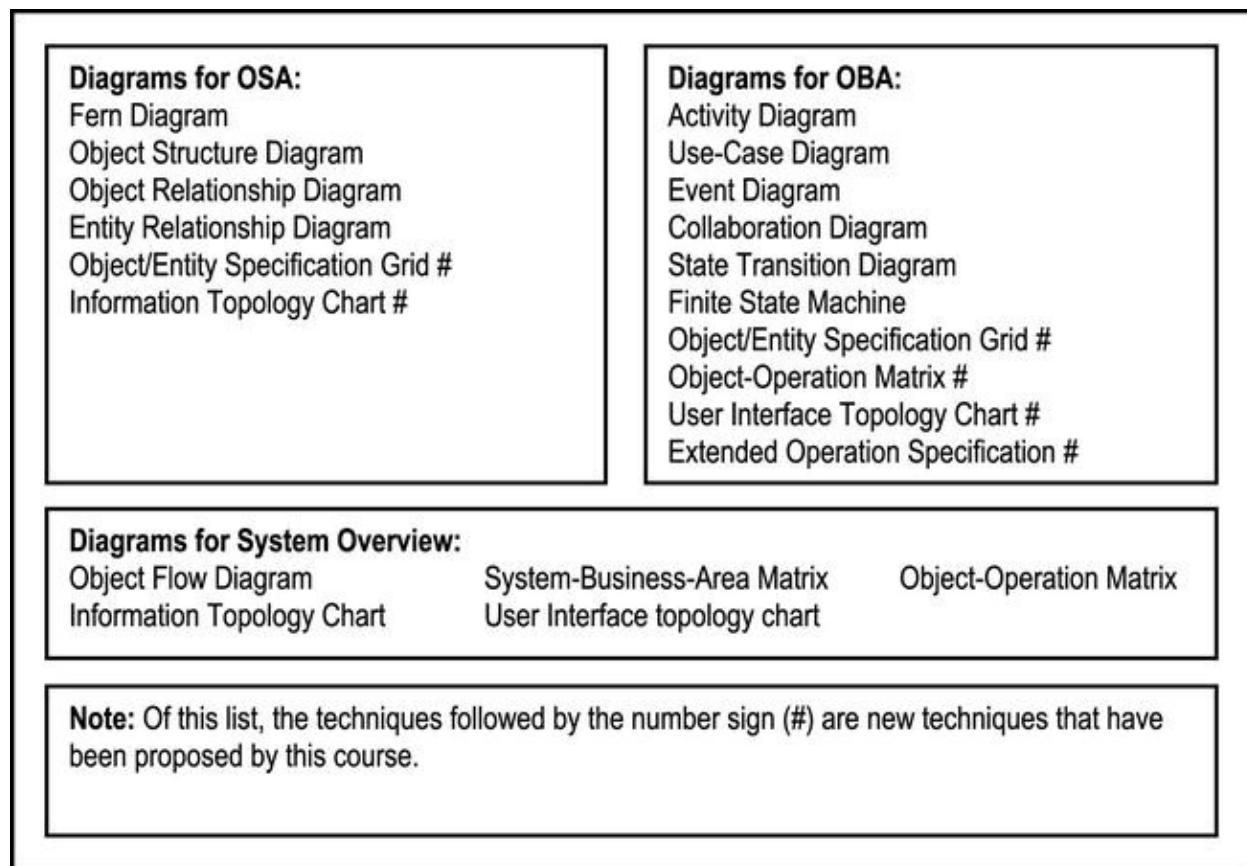


Figure A3-2. Recommended OOM Diagramming Techniques

A3.4 Enterprise Planning

A full discussion of enterprise planning is beyond the scope of this course; however, in the interest of comprehensive coverage, a brief summary is provided here. In the broadest sense, enterprise planning relates to planning the essentials of the organization. In OOIE, we concentrate on issues that relate (directly or indirectly) to the information infrastructure of the organization. As such, enterprise planning involves a number of important activities, including the following:

- Planning the information infrastructure of the organization
- Managing that infrastructure in the face of changing user requirements, changing technology, challenges from competitors, etc.

- Planning and pursuing appropriate IT strategies for the organization
- Having a proper backup and recovery policy
- Human and technical resource management
- Project management

Of course, each of these activities involves has its own set of principles and methodologies. Three very useful diagramming techniques here are the information topology chart (ITC), the user interface topology chart (UITC), and the object flow diagram (ODF). [Figure A3-3](#) illustrates a partial ITC for the CUAIS project; [Figure A3-4](#) illustrates a partial UITC for the project; and [Figure A3-5](#) shows an OFD for the project.

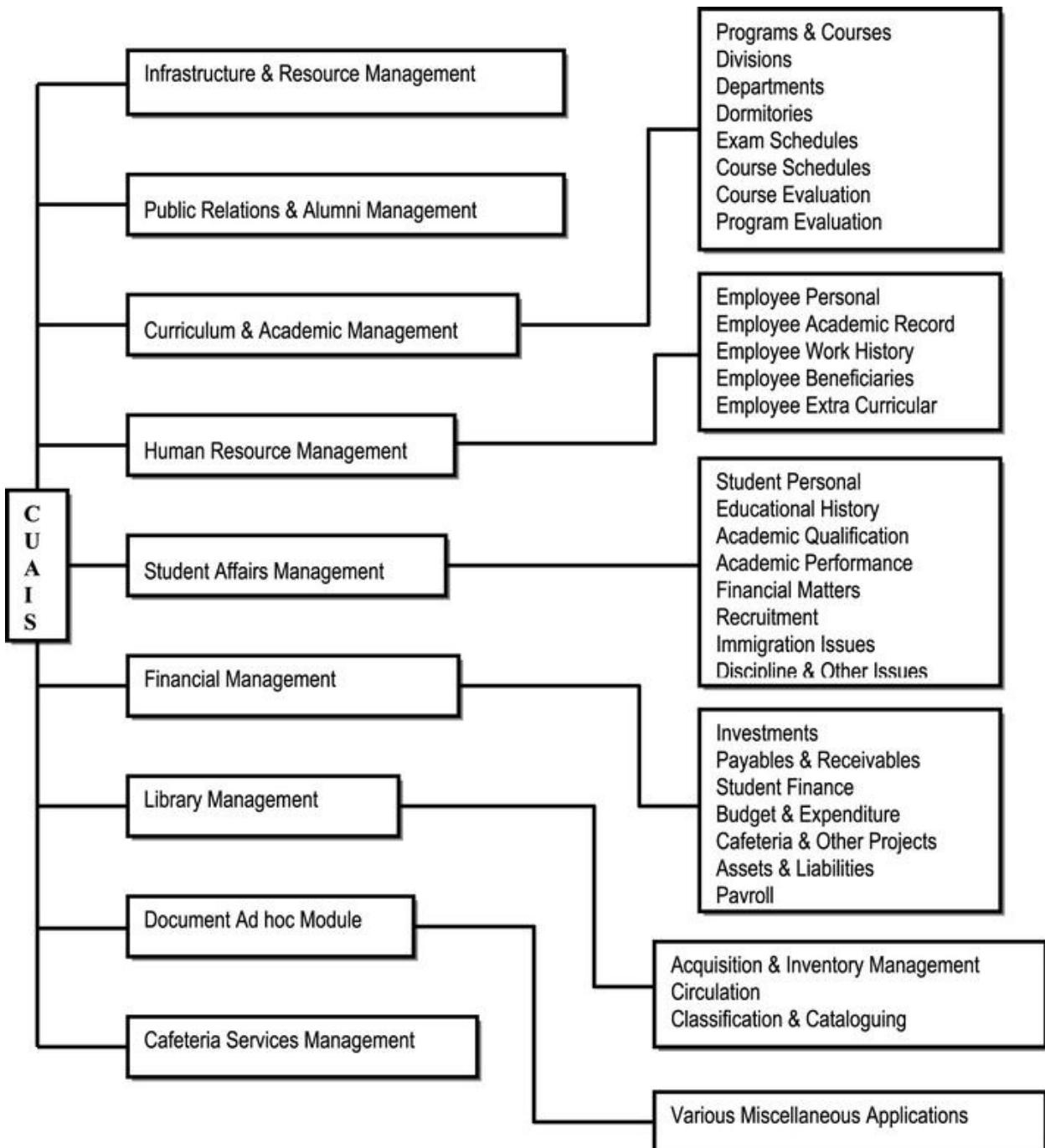
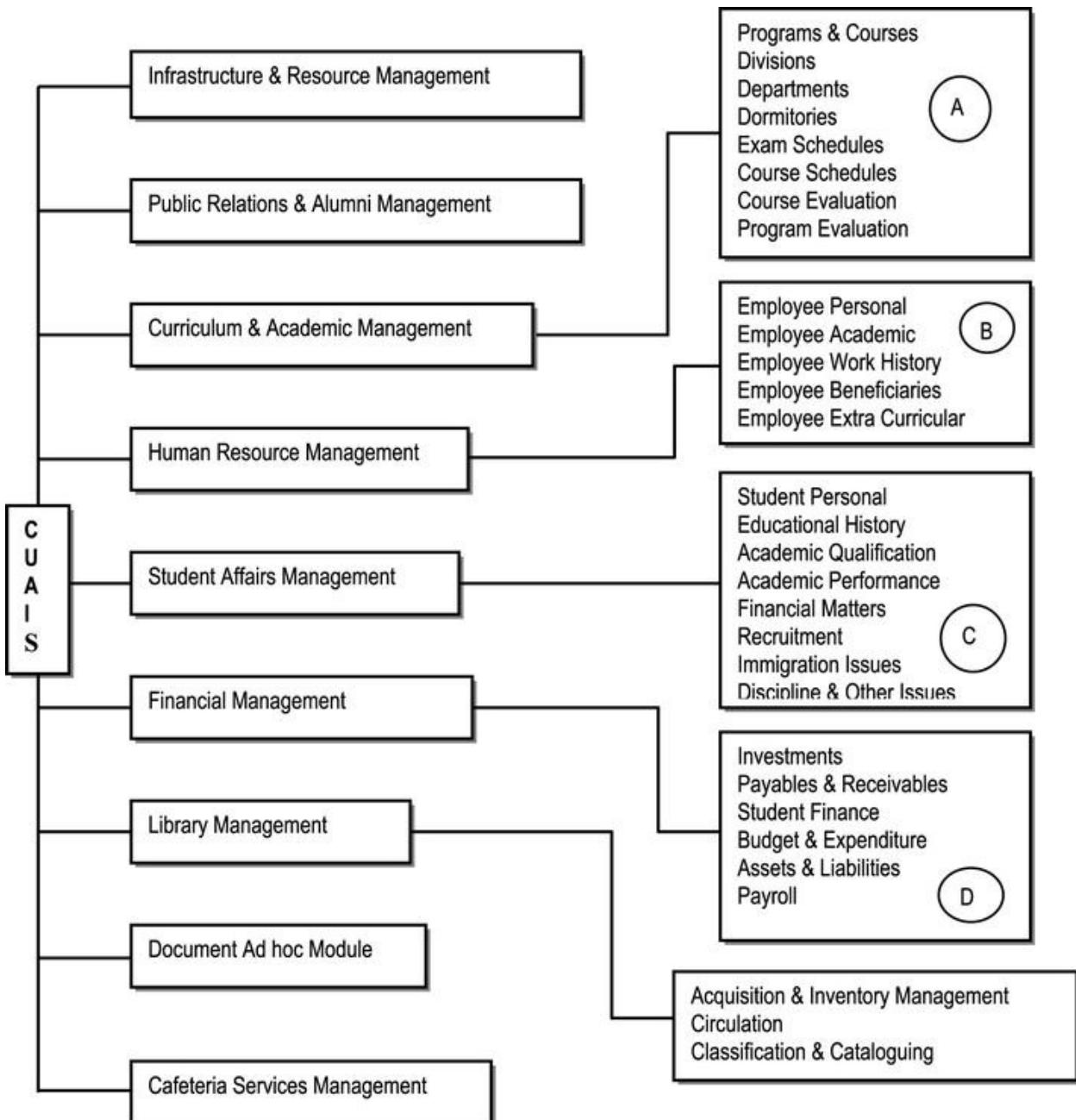


Figure A3-3. Partial Information Topology Chart for the CUAIS Project



A. Curriculum & Academic Management Menu

Add Academic Program
Modify Academic Programs
Remove Academic Programs
Inquire on Academic Programs
Report on Academic Programs
...
Add Program Evaluations
Modify Program Evaluations
Remove Program Evaluations
Inquire on Program Evaluations
Report on Program Evaluations

B. Human Resource Management Menu

Add Employees
Modify Employees
Remove Employees
Archive Employees
Inquire on Employees
Report on Employees
...

C. Student Affairs Management Menu

Add Students
Modify Students Personal
Record Students Academic
Record Students Non-academic
Modify Students Academic
Inquire on Students Non-academic
Inquire on Student Academic
Report on Students Non-academic
Report on Students Academic
...

D. Financial Management Menu

Add Investments
Modify Investments
Inquire on Investments
Report on Investments
...
Add Budget & Expenditure
Modify Budget & Expenditure
Inquire on Budget & Expenditure
Report on Budget & Expenditure

Figure A3-4. Partial UITC for the CUAIS Project

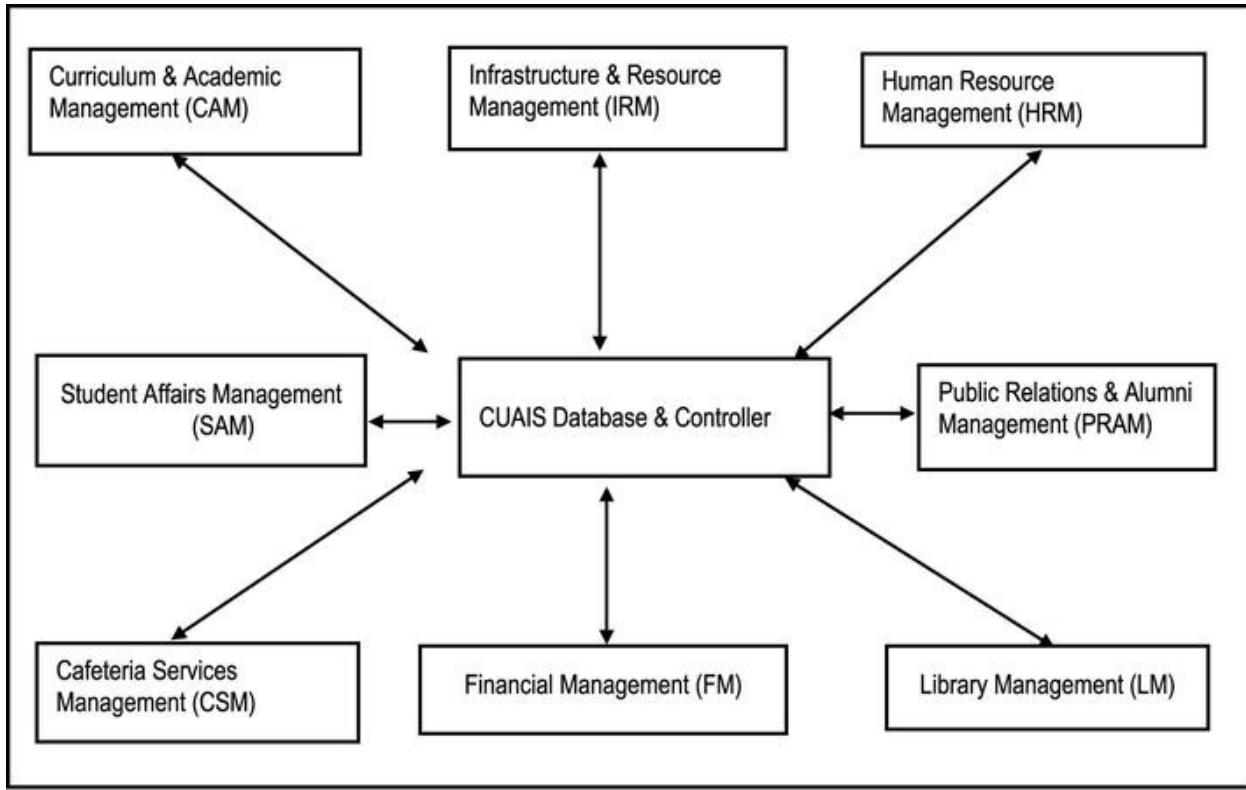


Figure A3-5. OFD for the CUAIS Project, Assuming Repository Model

A3.5 Business Area Analysis

In *business area analysis* (BAA), we are interested in the nature of participation that each business area of the organization will have (or already has) on the component systems comprising the information infrastructure. In particular the information system executive wants to ensure the following:

- The underlying object (database) structure must meet the needs of various business areas.
- The operations provided must meet the requirements of the various business areas.
- Appropriate security mechanism must be put in place to ensure the resources (objects and operations) of the integrated system are accessible to the relevant functional

areas of the organization.

Typically, diagrams and/or matrices are constructed to reflect critical information:

- Object structure diagrams indicate the structure and operational services of information object types (in case of object oriented (OO) approach). Alternately, you may use object/entity specification grids (O/ESGs). [Figure A3-6](#) provides an illustration of O/ESG for four of the object types (or entities) that would comprise a typical manufacturing firm's database. In actuality, there would be one for each object type (or entity) comprising the system.

O1 – Department [RMDepartment_BR]
Attributes: 01. Department Number [Dept#] [N4] 02. Department Name [DeptName] [A35] 03. Department Head Employee Number [DeptHeadEmp#] [N7] {Refers to O2.Emp#} ...
Comments: This table stores definitions of all departments in the organization.
Indexes: 1. Primary Key Index: RMDepartment_NX1 on [01]; constraint RMDepartment_PK. 2. RMDepartment_NX2 on [02]
Valid Operations: 1. Maintain Departments [RMDepartment_MO] 1.1 Add Departments [RMDepartment_AO] 1.2 Update Departments [RMDepartment_UO] 1.3 Delete Department [RMDepartment_ZO] 2. Inquire on Departments [RMDepartment_IQ]

O2 – Employee [RMEmployee_BR]**Attributes:**

01. Employee Identification Number [Emp#] [N7]
 02. Employee Last Name [EmpLName] [A20]
 03. Employee First Name [EmpFName] [A20]
 04. Employee Middle Initials [EmpMInitl] [A4]
 05. Employee Date of Birth [EmpDOB] [N8]
 06. Employee's Department [EmpDept#] [N4] {Refers to O1.Dept#}
 07. Employee Gender [EmpGender] [A1]
 08. Employee Marital Status [EmpMStatus] [A1]
 09. Employee Social Security Number [EmpSSN] [N10]
 10. Employee Home Telephone Number [EmpHomeTel] [A14]
 11. Employee Work Telephone Number [EmpWorkTel] [A10]
- ...

Comments:

This table stores standard information about all employees in the organization.

Indexes:

1. Primary Key Index: RMEmployee_NX1 on [01]; constraint RMEmployee_NX.
2. RMEmployee_NX2 on [02, 03, 04]
3. RMEmployee_NX3 on [09]

Valid Operations:

1. Manage Employees [RMEmployee_MO]
 - 1.1 Add Employees [RMEmployee_AO]
 - 1.2 Update Employees [RMEmployee_UO]
 - 1.3 Delete Employees [RMEmployee_ZO]
2. Inquire on Employees [RMEmployee_IO]
3. Report on Employees [RMEmployee_RO]

O3 – Supplier [RMSupplier_BR]
Attributes:
01. Supplier Number [Suppl#] [N4] 02. Supplier Name [SupplName] [A35] 03. Supplier Contact Name [SupplContact] [A35] 04. Supplier Telephone Numbers [SupplPhone] [A30] 05. Supplier E-mail Address [SuppEmail] [A30] ...
Comments:
This table stores definitions of all employee classifications.
Indexes:
1. Primary Key Index: RMSupplier_NX1 on [01]; constraint RMSupplier_PK. 2. RMSupplier_NX2 on [02] 3. RMSupplier_NX3 on [04]
Valid Operations:
1. Manage Suppliers[RMSupplier_MO] 1.1 Add Suppliers [RMSupplier_AO] 1.2 Update Suppliers [RMSupplier_UO] 1.3 Delete Suppliers [RMSupplier_ZO] 2. Inquire on Suppliers [RMSupplier_IO] 3. Report on Suppliers [RMSupplier_RO]
O4 – Project [RMPProject_BR]
Attributes:
01. Project Number [Proj#] [N4] 02. Project Name [ProjName] [A15] 03. Project Summary [ProjSumm] [M] 04. Project's Manager [ProjManagerEmp#] [N7] {References O2.Emp#} ...
Comments:
This table stores definitions of all company projects.
Indexes:
1. Primary Key Index: RMPProject_NX1 on [01]; constraint RMPProject_PK. 2. RM Project_NX2 on [02]
Valid Operations:
1. Manage Projects [RMPProject_MO] 1.1 Add Projects [RMPProject_AO] 1.2 Update Projects [RMPProject_UO] 1.3 Delete Projects [RMPProject_ZO] 2. Inquire on Projects [RMPProject_IO] 3. Report on Projects [RMPProject_RO]

Figure A3-6. Partial O/ESG for Manufacturing Environment

- Object-operation matrices show how various object types will be used in different business areas.
- System-component-business-area matrices show what functional areas will use certain system components of the information system, and how.

In constructing the matrices for the BAA, always cluster related components (object types or sub-systems) together. [Figure A3-7](#) illustrates an object-operation matrix while [Figure A3-8](#) illustrates a component-business-area matrix.

	Business Areas							
Object Type	Finance	Accounting	HRD	Marketing	Production	Engineering	Info. Tech	
Chart of Accounts	UQ	AUDQ	Q	Q	Q	Q	Q	
Transactions	UQ	AUDQ	Q	Q	Q	Q	Q	
Payments Received	Q	AUDQ	Q	AUD0	Q	Q	Q	
Payments Made	Q	AUDQ	Q	AUD0	Q	Q	Q	
Financial Institution	Q	AUDQ	Q	Q	Q	Q	Q	
Budget	AUDQ	AUDQ	AUDQ	AUDQ	AUDQ	AUDQ	AUDQ	
Customers	Q	AUDQ	Q	Q	Q	Q	Q	
Investments	AUDQ	AUDQ	Q	Q	Q	Q	Q	
Key:	A = Add	U = Update	D = Delete	Q = Query/Report				
Note:								
<ol style="list-style-type: none"> 1. The matrix tells what operations of each object type are applicable for different business areas, when this Financial Management System is implemented. 2. The information provided could also be used to help define the security arrangements for the system. 3. A matrix of this sort may be constructed for each component system making up the information infrastructure of the organization. 								

[Figure A3-7. Object-Operation Matrix for a Financial Management System](#)

Component Systems	Cafeteria	Finance & Planning	A Academic Admin.	PR & Marketing	Student Services	Academic Depts.	Human Resource	Information Technology
Financial Management	Q	AUDQ	Q	Q	Q	Q	Q	Q
Human Resource Management		Q	Q	Q	Q	Q	AUDQ	Q
Library Management		Q	AUDQ	Q	Q	Q		Q
Cafeteria Services Management	AUDQ	Q	Q	Q	Q	Q	Q	Q
Curriculum & Acad. Management		Q	AUDQ	Q	Q	Q	Q	Q
PR & Alumni Management		Q	Q	AUDQ	Q	Q	Q	Q
Student Affairs Management		Q	Q	Q	AUDQ	Q	Q	Q

Key: A = Add U = Update D = Delete Q = Query/Report

Note: The matrix tells how each business area will be able to access each component system comprising the information infrastructure.

Figure A3-8. Component-Business-Area Matrix for a University

A3.6 System Design

System design relates to design of the component systems that comprise the information infrastructure of the organization. To be excellent at designing information infrastructures, you need to have mastery of software engineering, database systems, and an understanding of how a typical business works. You also need to have a working knowledge of programming, and outstanding communication and leadership skills. You also need to have mastery of various software diagramming techniques, many of which have been covered in the course. [Figure A3-9](#) provides a checklist of design issues with which you should be familiar (review [chapters 9 – 13](#)).

Design Activity	Resultant Design Spec Component
Architectural Design	System Architecture Specification
Interface Design	System Interface Specification
Database (Object Structure) Design	Database (Object Structure) Specification
Operations Design	Operations Specification
User Interface Design	User Interface Specification
Documentation Design	System Documentation Specification
Message Design	Message Specification
Security Design	Security Specification

Figure A3-9. Summary of System Design Considerations

Following are some basic guidelines that should guide the design process:

- The design must be accurate and comprehensive, covering both structure and behavior of system components, and with due consideration to the availability of technology.
- The design must adequately take care of current and future requirements of the (component systems in the) organization.
- The design must be pragmatic.
- The design must be informed by the software quality factors emphasized throughout the course (review [chapters 1 and 3](#)).
- The design must uphold established software engineering standards.

A3.7 System Construction

System construction relates to the development of the component systems that comprise the information infrastructure of the organization. If enterprise planning, business area analysis and system design were accurate and comprehensive, system construction will simply be the implementation of a carefully developed plan. You will recall from your knowledge of software engineering, that system acquisition may take any of several alternatives. [Figure A3-10](#) summarizes these alternatives.

1. Traditional waterfall approach (in-house or via contracted work)
2. Prototyping (phased or rapid, in-house or via contracted work)
3. Iterative development (in-house or via contracted work)
4. Assembly from re-usable components (in-house or via contracted work)
5. Formal transformation (in-house or via contracted work)
6. Agile development (in-house or via contracted work)
7. Customizing an application software package
8. End-user development
9. Outsourcing

Figure A3-10. Software Acquisition Alternatives

Finally, please remember the following points:

1. Software system design and construction must conform to established software development standards.
2. Software acquisition must follow a clearly developed plan with specific targets (deadlines).
3. Prudent project management will be required if targets are to be met.

A3.8 Summary and Concluding Remarks

Let us summarize what we have covered in this chapter:

- Information engineering (IE) is the planning, construction and management of the entire information infrastructure of the organization. Object-oriented information engineering (OOIE) is IE conducted in an object-oriented manner.
- OOIE may be conducted by pursuing four steps: enterprise planning, business area analysis, system design, and system construction.
- Enterprise planning relates to planning the essentials of the organization. In OOIE, we concentrate on issues that relate (directly or indirectly) to the information infrastructure of

the organization.

- In business area analysis (BAA), we are interested in the nature of participation that each business area (of the organization) will have (or already has) on the component systems (and system components).
- System design relates to design of the component systems that comprise the information infrastructure of the organization.
- System construction relates to the development of the component systems that comprise the information infrastructure of the organization.
- In conducting OOIE, there must be mastery and appropriate use of various diagramming techniques that are applicable to OOSE.

Since its introduction, information engineering has been overtaken by (or more precisely, morphed into) methodologies such as *data administration*, and *business intelligence* (see [Turban, 2007] and [Turban, 2008]). However, the principles of IE are still relevant. When OOIE is combined with life cycle models such as phased prototyping, iterative development, agile development, or CBSE, you have formula for successful software engineering.

Once you have obtained a good overview of the information infrastructure of the enterprise, your next step is to zoom in on each component system and provide more detail in terms of the structure and behavior of constituent objects comprising the system. The next chapter will address the relevant issues relating to this.

A3.9 References and/or Recommended Reading

[Martin, 1989a] Martin, James and James Odell. *Information Engineering Book I: Introduction*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1989.

[Martin, 1989b] Martin, James and Joe Leben. *Strategic Information Planning*

Methodologies. 2nd ed. Eaglewood Cliffs, New Jersey: Prentice Hall, 1989. See [chapters 2 and 13](#).

[Martin, 1990a] Martin, James and James Odell. *Information Engineering Book II: Planning and Analysis*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1990.

[Martin, 1990b] Martin, James and James Odell. *Information Engineering Book III: Design and Construction*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1990.

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1993. See [chapter 5](#).

[Sprague, 1993] Sprague, Ralph H. and Barbara C. McNurlin *Information Systems Management in Practice*. 3ed Eaglewood Cliffs, New Jersey: Prentice Hall, 1993. See [chapters 4](#).

[Turban, 2007] Turban, Efraim, Jay Aaronson, Ting-peng Liang, and Ramesh Sharda. *Decision Support and Business Intelligence*. Eaglewood Cliffs, New Jersey: Prentice Hall, 2007.

[Turban, 2008] Turban, Efraim, Ramesh Sharda, Jay Aaronson, and David King. *Business Intelligence*. Eaglewood Cliffs, New Jersey: Prentice Hall, 2008.

APPENDIX 4



Basic Guidelines for Object-Oriented Methodologies

In this chapter, we shall discuss some of the basic guidelines for OOSE. The chapter proceeds under the following subtopics:

- Object Identification
- End-User Involvement
- OO Diagramming
- Enterprise-wide Design
- Emphasis on OO-CASE Tools versus OO-Programming Language
- OO Modeling
- Summary and Concluding Remarks

A4.1 Object Identification

Skilled analysis is required in identifying object types. It is very important to identify object types that will make up the software system. In traditional software engineering, the primary question that the software engineer used to ask is, what do you do? In contemporary OOSE, the primary question that the software engineer seeks to ask and obtain an answer for is, what information do you manage, and how?

From your software engineering fundamentals, you recall that there are

certain techniques of information gathering that are available to the software engineer, as he/she seeks to define the requirements of a software system (review [chapter 5](#)). Most of these techniques are applicable here; only, the focus is different. The most applicable techniques include the following:

- Analysis of Documents
- Interview
- Questionnaire
- Sampling and Observation
- Brainstorming
- Prototyping
- Observation

OOAD is not a replacement for good database design. In fact, for OOAD to succeed there must be good database design. As such, the principles of database design (learned in your database systems course) are not to be discarded. Rather, they form the basis for good OO design.

Several approaches to object identification (more precisely object type identification) have been proposed. The truth, however, is that these approaches do not offer any guarantee of successful identification of all object types required by a given system. Among the approaches that have been proposed are the following:

- Using Things to be Modeled
- Using Definitions of Objects, Categories and Types
- Using Decomposition
- Using Generalizations and Subclasses
- Using OO Domain Analysis or Application Framework
- Reusing Individual Hierarchies, Objects and Classes
- Using Personal Experience
- Using the Descriptive Narrative Approach
- Using Class-Responsibility-Collaboration Card

- Using the Rule-of-Thumb Approach

A4.1.1 Using Things to be Modeled

This is the preferred approach of experienced software engineers. It is summarized in two steps:

1. Identify tangible objects (more precisely, object types) in the application domain that are to be modeled.
2. Put these objects into logical categories. These categories will become super-classes.

In order for this method to be successful, the software engineer must make a paradigm shift to an object-oriented mindset. For this reason, software engineers who have not made that transition, usually have difficulties employing it.

The main drawback of this approach is that on the surface, it does not help in the identification of abstract (intangible) object types. Of course, the counter argument here is that with experience, identifying abstract object types will not be a problem.

A4.1.2 Using the Definitions of Objects, Categories and Interfaces

This technique assumes that the most effective way to identify object types is to do so directly, based on the software engineer's knowledge of the application domain, as well as his/her knowledge of and experience in object abstraction and object categorization. Objects, categories (classes) and interfaces are identified intuitively.

This approach is very effective for the experienced software engineer who has made the paradigm shift of OO methodologies.

A4.1.3 Using Decomposition

This approach assumes that the system will consist of several component objects, and works very well in situations where this assumption is true. The

steps involved are:

1. Identify the aggregate objects or categories.
2. Repeatedly decompose aggregate objects into their components until a stable state is reached.

There are two drawbacks with this approach:

- Not all systems have an abundance of aggregate object types.
- If the approach is strictly followed, one might end up proposing component relationships where one-to-many (or many-to-many) relationships would better serve the situation.

A4.1.4 Using Generalizations and Subclasses

This technique contends that objects are identified before their categories (object types). The steps involved are:

1. Identify all objects.
2. Identify objects that share the same attributes and services (operations). Generalize these objects into categories (object types).
3. Identify categories (classes) that share common resources (attributes, relationships, operations, etc.).
4. Factor out the common resources to form super-classes, then use generalization or specialization for all categories that share these common resources to form sub-categories (subclasses).
5. If the only common factors are service prototypes, use the *interface* to factor out these common factors.

The main advantage of this technique is reuse — it is likely to produce a design that is very compact, due to a high degree of code reuse.

The main disadvantage of the technique is that it could be easily misused to

produce a design of countless splinter classes that reuse logically unrelated resources. This could result in a system that is difficult to maintain.

A4.1.5 Using OO Domain Analysis or Application Framework

This approach assumes that an OO domain analysis (OODA) and/or application framework of an application in the same problem domain was previously done, and can therefore be used. The steps involved are:

1. Analyze the existing OODA or application framework for the problem.
2. Reuse (with modifications where necessary) objects and/or categories as required.

The main advantage of this approach is that if such reusable components can be identified, system development time can be significantly reduced.

The main drawback of the approach is that it is not always applicable. Most existing systems either have incomplete OODA or no OODA at all. This should not be surprising since software engineering is a fairly youthful discipline. With time, this approach should be quite useful.

A4.1.6 Reusing Hierarchies, Individual Objects and Classes

This technique is relevant in a situation where there is a repository with reusable class hierarchies, which is available. The steps involved are:

1. Use the repository to identify classes and class hierarchies that can be reused.
2. Where necessary, adopt and modify existing classes to be reused.
3. Introduce new classes (categories) where necessary.
4. If the classes are parameterized, supply the generic formal

parameters (this is not currently supported in Java).

The advantages and disadvantages of this approach are identical to those specified in the previous sub-section.

A4.1.7 Using Personal Experience

This approach is likely to become more popular as software engineers become more experienced in designing software via object technology. The software engineer draws on his/her experience in previously designed systems, to design a new one. The required steps are:

1. Identify objects, categories (classes) and interfaces that correspond to ones used in previous models that are in the same application domain.
2. Utilize these resources (with modification where applicable) in the new system.
3. Introduce new categories (object types) and interfaces where necessary.

This approach could significantly reduce the development time of a new system, while building the experience repertoire of the software engineer(s) involved.

Where the relevant prior experience is lacking, this approach breaks down and becomes potentially dangerous. Also, the approach could facilitate the proliferation of shoddy design with poor documentation, or no documentation at all. This could result in a system that is very difficult to maintain.

A4.1.8 Using the Descriptive Narrative Approach

The descriptive narrative technique, pioneered by Russell J. Abbott and popularized by Grady Booch, was widely used in the mid-1980s. It may be summarized as follows:

1. Prepare or obtain a descriptive narrative of the requirements of the system.
2. From the narrative, identify nouns, pronouns and verbs.
3. Nouns and pronouns are used to identify objects, and object types (categories), while verbs are used to identify operations (services) that are to be defined on the object types.

[Figure A4-1](#) provides an illustration of a descriptive narrative, while [Figure A4-2](#) shows what object types and operations can be determined from the narrative. Further refinement is left as an exercise.

Purchase orders are sent to suppliers, requesting inventory items in specific quantities. If a PO is incorrectly generated, it is immediately removed and a new PO generated. The purchase invoice is the official document used to recognize receipt of goods from suppliers. All goods received are accompanied by invoices. Once received, the invoice is recorded. Items received are also recorded and appropriate inventory adjustments made to the inventory item master file. Receipt quantities can be adjusted, but if wrong items are recorded on receipt, or omissions are made, the whole invoice must be removed and re-recorded. When a receipt is correctly recorded, the associated PO status is adjusted.

Figure A4-1. Descriptive Narrative for Purchase Order and Invoice Receipt Subsystem

Object Type	Data Description	Operations
Purchase Order	Stores Order Number, Order Date, Supplier, Items Ordered and related Quantity Ordered, etc.	Generate, Remove, Adjust-Status
Supplier	Stores Supplier Code, Supplier Name, Supplier Address, Contact Person, Telephone, E-mail, etc.	Sent-Invoice
Purchase Invoice	Stores Invoice Number, Invoice Date, Related Supplier, Items Shipped and related Quantity Shipped, Invoice Amount, Discount, Tax, etc.	Record, Remove, Adjust-Quantity
Inventory Item	Stores Item Code, Item Name, Item Category, Quantity on Hand, Last Purchase Price, etc.	Adjust-Inventory

Figure A4-2. Object Types and Operations for Purchase Order and Invoice Receipt Subsystem

The main advantages of the technique are that it is easy to learn, understand and master; also, it is applicable to virtually any situation.

The technique has two main shortcomings:

- It may not always yield the correct set of object types, since nouns do not always translate to object types (classes). For example, the phrase “commit the transaction” contains a noun (“transaction”) and a verb (“commit”), but it would be

foolhardy to model “transaction” as an object type, and “commit” as an operation.

- As the size and complexity of the system increases, the descriptive narrative becomes more voluminous. Conceivably, we could get to a point where in order to identify object types one has to first write several pages of narrative.

A4.1.9 Using the Class-Responsibility-Collaboration Method

A popular concept in object identification is *responsibility-driven design*, a term used to mean, we identify classes and determine their responsibilities. This must be done well before the internals of the class can be tackled.

The *CRC* (*class-responsibility-collaborator*) methodology, proposed by Kent Beck and Ward Cunningham (see [Beck & Cunningham, 1989]), defines for each class, its responsibilities and collaborators that the class may use to achieve its objectives (hence the acronym). The responsibilities of a class are the requests it must correctly respond to; the collaborators of a class are other classes it must invoke in order to carry out its responsibilities. In carrying out its responsibilities, a class may use its own internal methods, or it may solicit help via a collaborator’s method(s).

The CRC methodology is summarized in the following steps:

1. In a brainstorming session, identify object types (categories) that may be required for the system.
2. For each object type (category), construct a list of (possible) services to be provided (or operations to be performed on an instance of that object type).
3. For each object type (category), identify possible collaborators.
4. Identify missing object types (categories) and interfaces that should be added. To identify new object types, look for attributes and services that cannot be allocated to the current set of object types. To identify new interfaces,

look for common service protocols.

5. Develop a CRC card for each object type and place them on a whiteboard (or desk surface). Draw association arcs from object types (classes) to collaborators.
6. Test and refine the model by utilizing *use cases*.

[Figure A4-3](#) provides an illustration of a CRC card. The CRC card must be stored electronically as well as physically, to aid the design process. For instance, CRC cards can be easily classified, printed, and strung out on a table, during a brainstorming session, to assist designers with gaining a comprehensive overview of the system. Class names and responsibilities must be carefully worded to avoid ambiguities.

Class Name	
Responsibilities	Collaborators
• • •	• • •

[Figure A4-3](#). The CRC Card

The main advantages of the technique are:

- It is easy to learn and follow.
- When used properly, it helps designers gain a comprehensive overview of the system.

The main drawback of the technique is that it requires experienced software engineer(s) if success is to be achieved.

A4.1.10 Using the Rule-of-Thumb Method

This technique has been proposed by the current author, after having successfully tested it on several medium sized and large software systems over several years. The technique recommends identification of information entities (or object types) by applying appropriate information gathering techniques

(review [chapter 5](#)). Once these object types have been identified, the technique then recommends an intuitive approach for identification of required operations for each object, based on the observation that in many situations, the essential operations required for an object type can be anticipated. Most objects (data entities) that make up a software system will be subject to any combination of the following basic operations:

- Addition of data items
- Update of existing data items
- Deletion of existing data items
- Inquiry and/or analysis on existing information
- Reporting and/or analysis of existing information
- Retrieval of existing data
- Prediction of future data based on analysis of existing data

Obviously, not all operations will apply for all object types; also, some object types may require additional operations. The software engineer makes intelligent decisions about these exceptions, depending on the situation. Additionally, the level of complexity of each operation will depend to some extent on the object type.

In a truly OO environment, these operations may be included as part of the object's services. In a hybrid environment, the information entities may be implemented as part of a relational database, and the operations would be implemented as part of the superimposed user interface objects (consisting of windows, forms, etc.).

Like the other techniques, this rule-of-thumb approach does not guarantee success in identifying and outlining all the required object types for a software system in every scenario. However, it does guarantee a solid starting point toward achievement of this objective in many (perhaps most) scenarios.

A4.2 End User Involvement

OOSE must thoroughly involve end-users to ensure their satisfaction and acceptance. The software engineer or IT expert should not operate in a manner that is oblivious of the end-user. Rather, his/her role is to extract information

needs from the end-users and model it in a way that the users understand and are satisfied with. Three types of workshops are common:

- JEM – Joint Enterprise Modeling
- JRP – Joint Requirements Planning
- JAD – Joint Application Design

Each session is guided by a facilitator (usually an IT professional, skilled in OOM). Key end-users do most of the talking. The facilitator does the modeling for end-users to see and accept or revise. The facilitator should be well experienced in system and software design, an excellent motivator and communicator; must have a good reputation that inspires confidence; a skilled negotiator.

Typical questions asked by the facilitator about object types:

- What are the data attributes?
- What operations are to be defined on this object type?
- What business rules apply here?
- What are the responsibilities of this object?
- What is needed to improve decisions?
- What control decisions could affect this operation?
- Could this operation be eliminated?
- Could this step be delegated to another object?
- Should this decision be made in a different place?

Martin makes a number of specific recommendations about workshops: Workshops typically last for no longer than a week at-a-time, and should not be allowed to spread out over an extended period of time. Nothing should be allowed to stall or slow down the progress of the workshop. Issues that cannot be resolved in a reasonable timeframe must be declared open (by the facilitator), and subject to subsequent analysis and/or discussion leading to satisfactory resolution (see [Martin, 1993]).

In order for a user workshop to be successful, thorough preparation must take place. The software engineers on the project must do their homework in obtaining relevant information, analyzing it, and preparing working models

and/or proposal that will be examined and/or used in the workshop. Additionally, training of participants may be required prior to the workshop, in order to achieve optimum benefits. The training sessions should be carefully planned and administered. Here, the participants should be briefed on the expectations and activities of the workshops.

A4.3 OO Diagramming

The previous chapter (section A3.3) mentioned the diagrams that are used in OOM. For ease of reference, they have been repeated in [Figure A4-4](#). For all diagrams, the following guidelines apply:

- Ideally, OO diagrams should be executable, i.e., it should be possible to generate code from the diagram (this is possibly via using an OO-CASE tool).
- Diagramming standards must be easy to learn.
- Diagrams should be easy to understand by software engineers as well as end users.

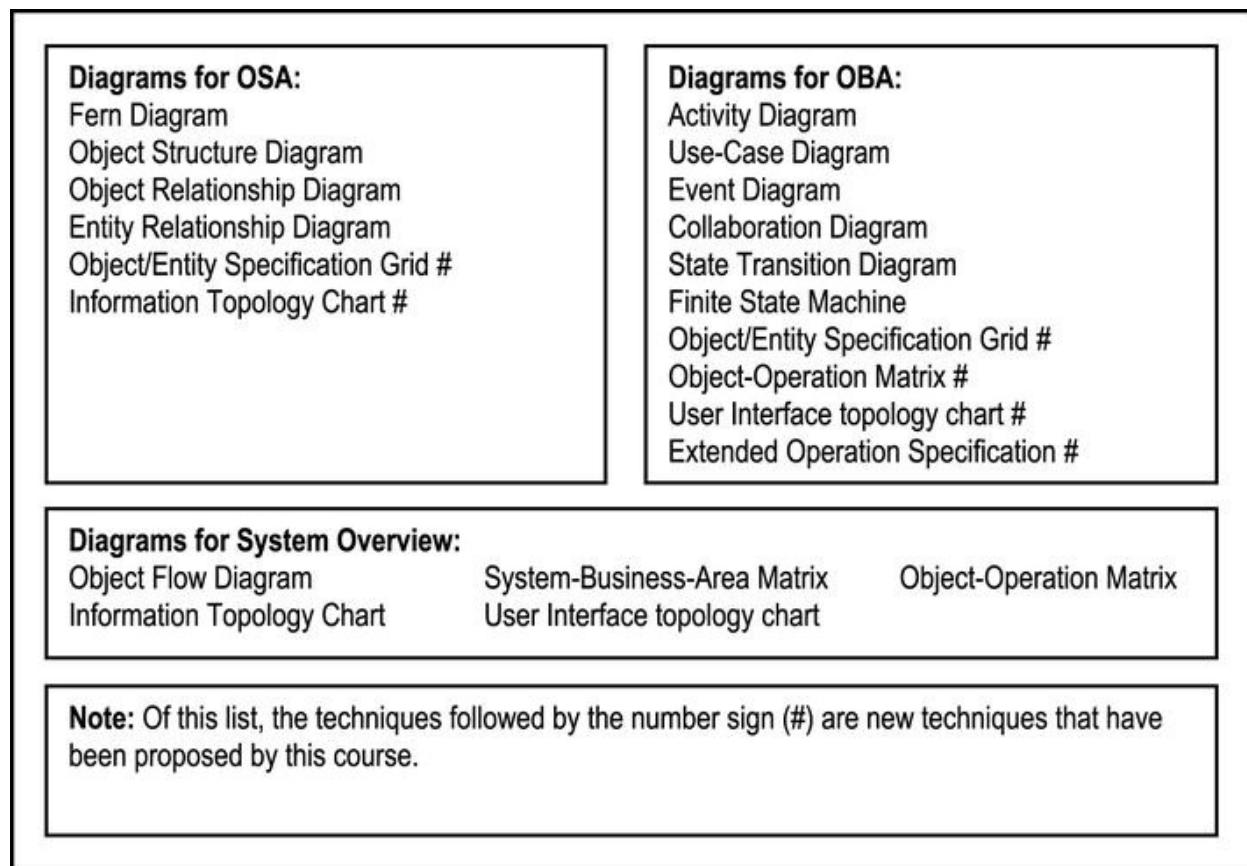


Figure A4-4. OOM Diagramming Techniques

A4.4 Enterprise-wide Design

Maximum benefits occur when analysis is done throughout the enterprise. OO information engineering (discussed in the previous chapter) enables a high-level overview model of the enterprise.

Enterprise-wide modeling has the following benefits:

- It leads to a more integrated information system.
- It leads to a higher level of reusability of code in the emergent system.
- It is a vital tool in enterprise redesign and reengineering.

In order to succeed, enterprise-wide design must be initiated by someone in authority, who commands the respect of the organization. In the early days, it was felt that enterprise-wide design had to be top-down, and never bottom-up.

However, with the passing of time, the industry has come to understand that for optimum effectiveness, top-down approaches should be combined with bottom-up approaches. Top-down provides the management clout and support needed; bottom-up provides the information support that is absolutely necessary.

A4.5 Emphasis on OO-CASE Tools versus OO-Programming Language

In 1993, James Martin predicted that for object technology to succeed, the thrust must be on OO-CASE tools, which generate (preferably platform independent) code, instead of traditional HLL coding [Martin, 1993]. We have seen the fulfillment of this prediction. Writing code via traditional high-level language (HLL) still has its place, but if this is the sole strategy for software construction, the demands of the user (mentioned in [appendix 1](#)) will never be met.

The Java revolution, along with products such as the Rational product line, offer significant promise in this area. Other prominent products include Delphi, C++ Builder, WebSphere, etc. ([appendix 7](#) will discuss OO software development tools).

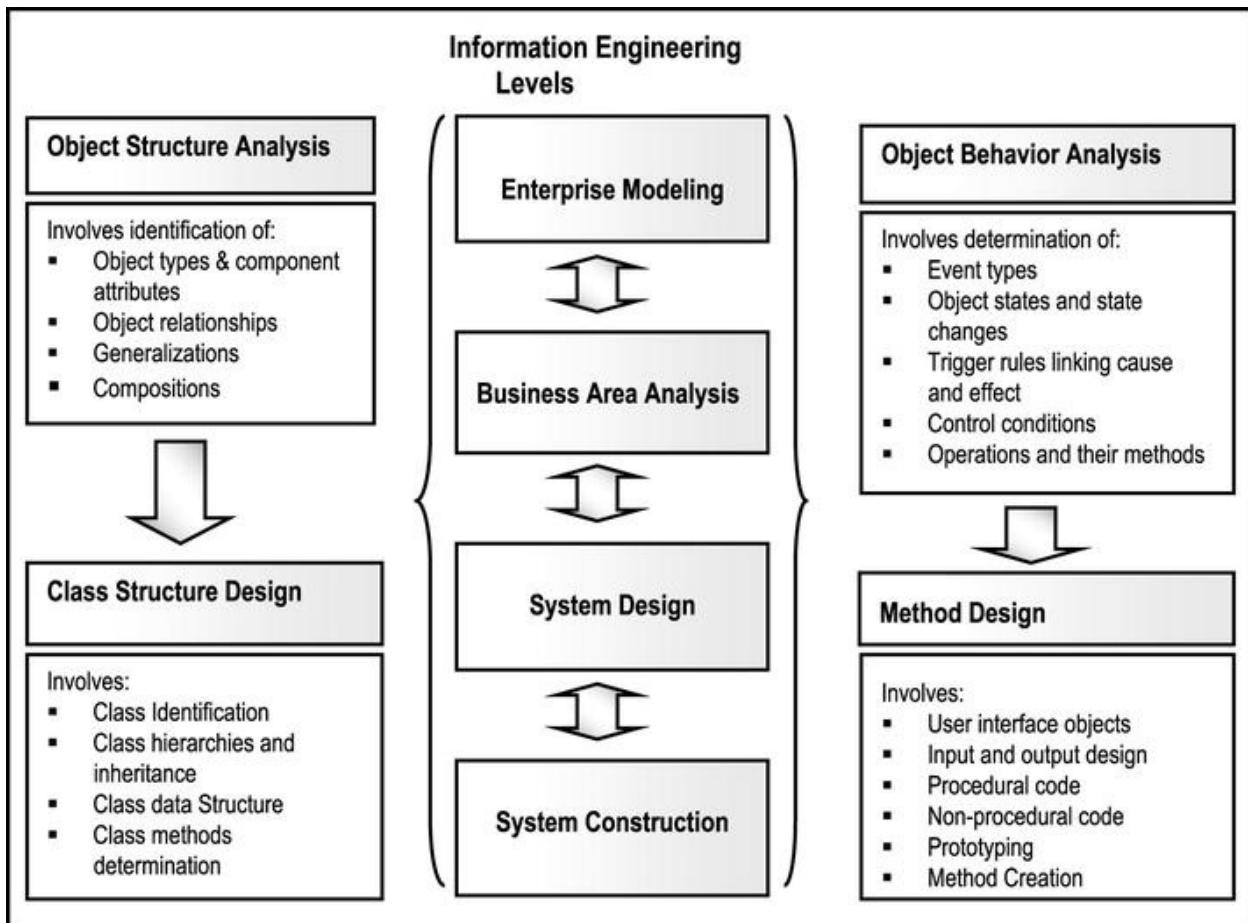
Assuming the use of CASE tools over traditional programming for future software development, the following guidelines apply:

- The code generated by CASE tools must be accessible: A software engineer or programmer must be able to access the code and modify it to introduce additional functionality to the software product being developed.
- The code generated by CASE tools must be portable and preferably platform independent.
- CASE tools should preferably facilitate the creation and maintenance of a repository, which stores information on all previously defined generic classes and methods. This will enhance code reusability.
- With repository-based development, paper is no longer the primary form of documentation.
- With OO-CASE tools, software development time may be

reduced by 20% – 60%.

A4.6 OO Modeling

As mentioned earlier, OO modeling has two aspects, which may be examined separately — structure and behavior. [Figure A4-5](#) illustrates an OO modeling hierarchy based on Martin's OO modeling pyramid. The figure shows how these two aspects relate to the OO modeling paradigm [Martin, 1993]. Here, two sides of the IS pyramid of [appendix 3](#) are emphasized — object structure and object behavior.



[Figure A4-5. OO Modeling Hierarchy](#)

Object structure analysis (OSA) utilizes diagrams that convey object types, object associates, compositions and generalization. Correspondingly, class structure design includes class identification, inheritance and data structure.

Object behavior analysis (OBA) utilizes object flow diagrams, state diagrams, event diagrams, and operation specifications. Correspondingly, method design includes operation identification and method creation.

In the traditional approach to software system acquisition, there are distinct, disjoint phases. Different diagrams and standards are used at each phase. Object technology uses one consistent model that is revisable. One set of diagrams and standards is employed. Consequently, the risk of having design errors is significantly reduced.

Again referring to [Figure A4-4](#), notice that the figure lists the diagramming techniques that are best suited for OSA, OBA, and system overview respectively. Several of these techniques are based on the UML notation, and are very easy to learn. Moreover, most contemporary software planning and development tools support UML. Once these techniques are learned and mastered (through experience), software design and construction (particularly the object-oriented way) becomes an exciting and fulfilling discipline that affects the life of the practicing software engineer, as well as countless others.

A4.7 Summary and Concluding Remarks

Here is a summary of what we have covered in this chapter:

- Object identification is critical to defining the structure of the system being designed. Among the various object identification techniques that have been proposed are the following: using things to be modeled; using definitions of objects, categories and types; using decomposition; using generalizations and subclasses; using OO domain analysis or application framework; reusing individual hierarchies, objects and classes; using personal experience; using nouns and verbs; using class-responsibility-collaboration cards; using the rule-of-thumb method.
- OOAD must thoroughly involve end-users to ensure their satisfaction and acceptance. Three types of workshops that are common are JEM (Joint Enterprise Modeling), JRP

(Joint Requirements Planning), and JAD (Joint Application Design).

- Mastery of OOM diagramming techniques is paramount.
- System design must be enterprise-wide within the context of OOIE.
- In constructing contemporary software, emphasis should be placed on OO-CASE tools rather than traditional HLLs. Sole reliance on the latter will not facilitate meeting the demands of the users.
- OO modeling involves two aspects — object structure analysis (OSA) and object behavior analysis (OBA). OSA relates to how the object types are comprised and their interrelatedness. OBA relates to the analysis and design of operations defined on each object type.

To proceed with our OO approach to software engineering, we need to delve deeper in the matter of OSA and OBA. The next chapter discusses the former.

A4.8 References and/or Recommended Reading

[Beck & Cunningham, 1989] Beck, Kent and Ward Cunningham. “A Laboratory for Teaching Object-Oriented Thinking.” *OOPSLA'89 Conference Proceedings October 1-6, 1989, New Orleans, Louisiana.*

<http://c2.com/doc/oops1a89/paper.html>.

[Due', 2002] Due', Richard T. *Mentoring Object Technology Projects*. Saddle River, New Jersey: Prentice Hall, 2002. See [chapters 3-6](#).

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development With UML and Java*. Upper Saddle River, NJ: Prentice Hall, 2002. See [chapter 4](#).

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1993. See

chapters 4 and 5.

APPENDIX 5



Categorizing Objects

This chapter focuses on how we categorize objects. As we work towards constructing a software system via the object-oriented paradigm, identifying the object types and their interrelationships becomes very crucial.

The chapter includes:

- Identifying Object Relationships
- Fern Diagram
- Information Topology Chart
- Object Relationship Diagrams
- Representing Details About Object Types
- Avoiding Multiple Inheritance Relationships
- Top-Down versus Bottom-Up
- Summary and Concluding Remarks

A5.1 Identifying Object Relationships

To identify relationships, you have to know what a relationship is, and what types of relationships there are. In the OO paradigm, we define a relationship as an association among two or more object types. There are seven types of relationships:

- One-to-one (1:1) relationship
- One-to-many (1:M) relationship

- Many-to-one (M:1) relationship
- Many-to-many (M:M) relationship
- Aggregation relationship
- Component relationship
- Inheritance (generalization) relationship

The first four types of relationships are referred to as *traditional relationships* because up until the OO model for database design gained preeminence, they were essentially the kinds of relationships that were facilitated by the relational database model. In the OO paradigm, they are referred to as *links*. Observe also, that the only difference between a 1:M relationship and a M:1 relation is a matter of perspective; thus, a 1:M relationship may also be described as a M:1 relationship (so that in practice, there are really three types of relationships). Put another way:

If O₁, O₂ are two object types and there is a 1:M relationship between O₁ and O₂, an alternate way of describing this situation is to say that there is a M:1 relationship between O₂ and O₁.

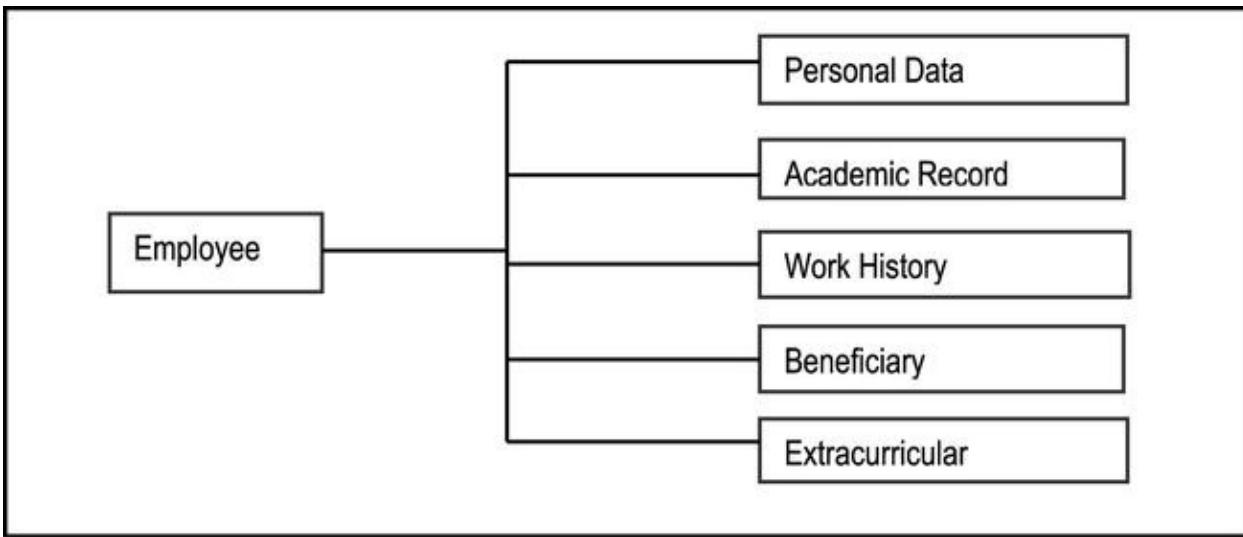
Your course in database systems allows you to gain mastery in how these relationships are represented and implemented in the relational model for databases. If you have not taken that course, you should plan on doing so in the near future. If you have already taken the course and need to review, please do so. [Chapter 10](#) of this course also provides an overview of relationship identification (albeit from a database perspective). The rest of this review focuses on some other techniques used in the OO paradigm.

A5.2 Fern Diagram

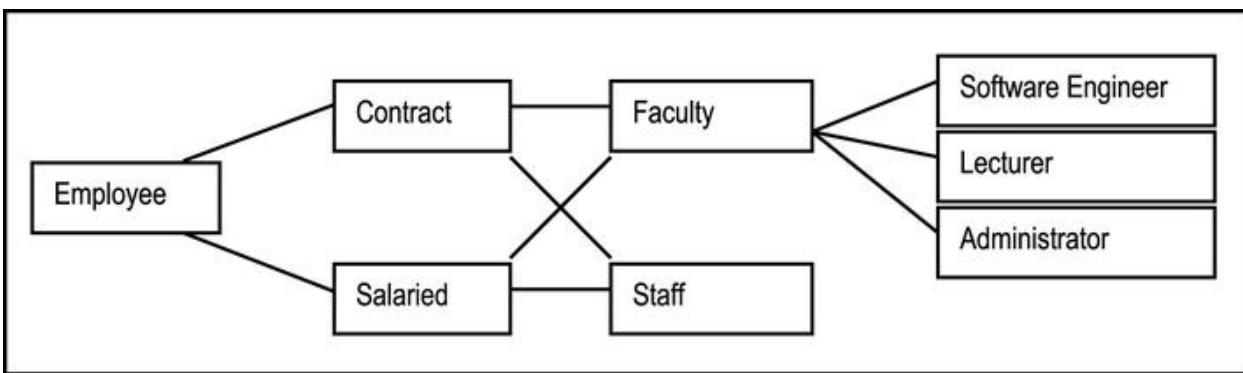
Fern diagrams are useful in depicting the object types that make up the system. A fern diagram may be tree structured (where there is no multiple inheritance) or network structured (where there is multiple inheritance). It typically includes aggregation, component and inheritance relationships but makes no distinction among them.

The fern diagram is usually read from left to right or top to bottom (no arrowheads required). It is a useful technique, particularly where system is large

and complex. [Figures A5-1](#) and [A5-2](#) provide two illustrations.



[Figure A5-1.](#) A Tree Structured Fern Diagram



[Figure A5-2.](#) A Network Structured Fern Diagram

Advantages of fern diagrams:

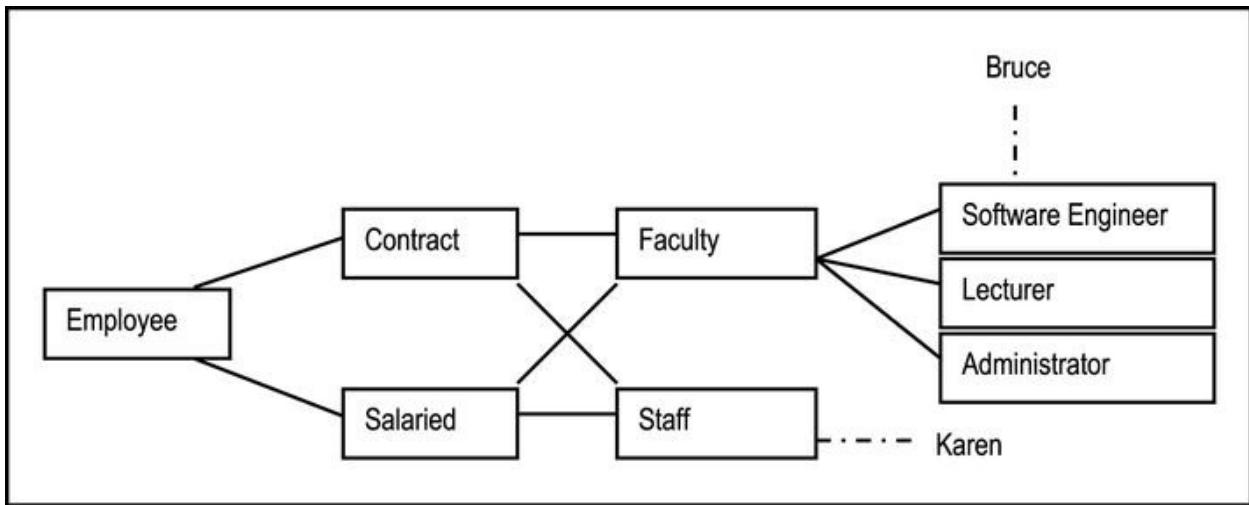
- They are easy to draw and maintain.
- They are useful in assisting in the categorization of objects.

The main drawbacks of the fern diagram are:

- No distinction is made between a subtype relationship and an aggregation relationship.
- Neither does it show other types of relationships.
- The diagram may become cluttered and unwieldy as the

system's size and complexity increases.

Sometimes it is useful to show instances on a fern diagram. This is done with the aid of broken lines. In the main, showing instances is impractical, but there are times when instances may have particular meaning in the design as in [Figure A5-3](#), where **Bruce** is a software engineer and **Karen** is a staff member.



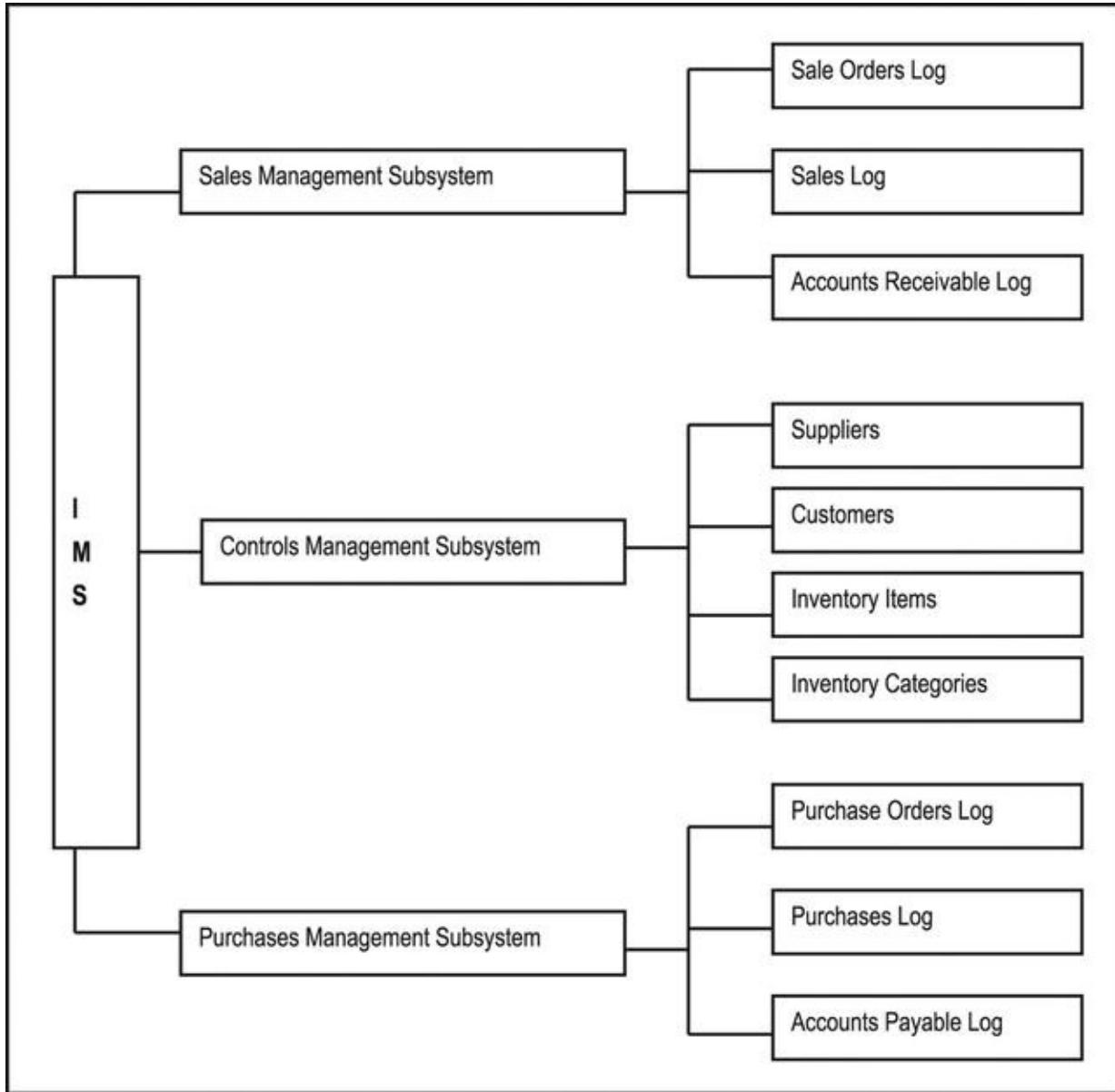
[Figure A5-3](#). Fern Diagram with Instances

A5.3 Information Topology Chart

The information topology chart (ITC) is a conceptual representation of component subsystems and object types (or information entities) of the software system. The ITC shows information levels of the software system in a top-down manner. The system name appears at the highest level. Subsystems appear at the next level, followed by information entities (or object types). If the software system contains no subsystems, then the information entities (or object types) appear at the second level. Optionally, data elements (attributes) may be included at the lowest level. The technique presents information to be managed in the system in a logical and modular way, and therefore allows for easy analysis and identification of omissions or redundancies.

The ITC is particularly useful in providing a global view of the system, including all significant components. It is useful for analysis and specification, as well as the design phases of the SDLC. As such, it illustrates component relationships, but no attempt is made at representing inheritance, or other types

of relationships. [Figure A5-4](#) illustrates a portion of an ITC for the generic Inventory Management System (IMS) project or earlier mention. A more detailed diagram appears in [appendix 10](#).



[Figure A5-4.](#) Partial Information Topology Chart for an Inventory Management System

At a cursory glance, you may be tempted to compare the ITC with the fern diagram. Here is the difference: The fern diagram is used to illustrate object categorization (including component and inheritance relationships). In contrast, the ITC's primary purpose is to illustrate how information will be classified and managed in the software system. As such, its focus is comprehensive coverage

of all object types (or information entities) for the software system being modeled. Component relationships are typically covered but that is not the primary focus of the technique. The ITC also differs from the HIPO chart (of [chapter 6](#)) in that whereas the HIPO chart is a functional representation of processes in traditional software systems, the ITC has a different focus as explained above.

Advantages of the ITC are as follows:

- The ITC is not just applicable in OOAD, but can also be used in function-oriented design (FOD).
- It is easy to draw and maintain.
- It is very effective in presenting a comprehensive perspective of the system.
- It is a powerful analysis and documentation tool.

Two limitations of the technique are worth noting:

- In presenting component information entities (object types), aggregation relationships are often incidentally represented. However, no distinction is made between subtype relationships and aggregation relationships.
- By intent, the technique does not show other types of relationships.

A5.4 Object Relationship Diagrams

An object-relationship diagram (ORD) is similar to an entity-relationship diagram (ERD). The conventions used in both techniques are also similar for the most part. However, there are a few subtle differences relating to how relationships are represented. Please review [chapter 10](#) (section 10.2.5) to refresh your memory on these differences.

OOAD does not replace good relational database design; rather it presumes it. There are, however, times when the two tend to contradict. In such cases, experienced judgment is required. It should be noted however, that the scenario of an OO-GUI being superimposed on top of a relational database is a widely

popular configuration.

It may become necessary to introduce abstract object types (classes) in the design, in the interest of generalization. An abstract object type is an object type that has no direct instances but whose descendants have direct instances. Artificially introduced abstract types are normally used as a mechanism for promoting code reuse. This in essence is the Principle of Occam's Razor: "two types should not be used where one will suffice" (see [Martin, 1993]).

The preferred standard for ORDs is the UML notation as described in [chapters 9 and 10](#) (sections 9.2.2 and 10.2.5). This notation is supported in many contemporary OO-CASE tools. Where an OO-CASE tool is not readily available, drawing on ORD becomes impractical for large, complex systems. Every effort should be made to have an ORD for the software system. However, if this is infeasible due to limited resources, alternate means should be explored. One alternative that was discussed in [chapter 10](#) and illustrated in [appendix 3](#) is the object/entity specification grid (O/ESG).

A5.5 Representing Details about Object Types

Two standard methodologies for representing details about object types are the object structure diagram (OSD) and the class-responsibility-collaboration (CRC) card. A third approach is the O/ESG, which was discussed in [chapter 10](#) (section 10.2.6) and illustrated in [appendix 3](#). A fourth possibility is to extend the information topology chart (ITC) of earlier discussions to include attributes of each object type (or information entity); if incorporated into an OO-CASE tool, this could be quite useful. We shall briefly discuss the two standard approaches.

A5.5.1 Object Structure Diagram

The object structure diagram (OSD) is just an alternate term for the class diagram, so you have already been introduced to it from your object-oriented programming, and the review provided in [chapter 9](#) (section 9.2.2). The recommended standard for OSDs is the UML notation. Typically, you will not find stand-alone OSDs for each object type comprising a system; rather, OSDs are incorporated in ORDs in order to convey useful information about the

structure and interrelatedness of object types comprising a software system. Note however, that from time to time, it might be necessary to highlight the OSD for a set of object types. One case in point would be where a software engineer is desirous of writing or modifying code for a specific set of object types. In situations where you are modeling a database, alternate methodologies such as ERDs and/or O/ESGs may be considered.

OSDs and ORDs (via the UML notation) are widely supported in contemporary software planning and development tools (review section 2.4.5 of [chapter 2](#)). The technique itself is quite simple and easy to follow. [Figure A5-5](#) shows an excerpt of the ORD for the CUAIS project of earlier mention (copied from [Figure 9-3](#) for ease of reference), depicting an inheritance relationship between object type **CollegeMember** (the super-type) on the one hand, and object types **Employee** and **Student** (the subtypes) on the other.

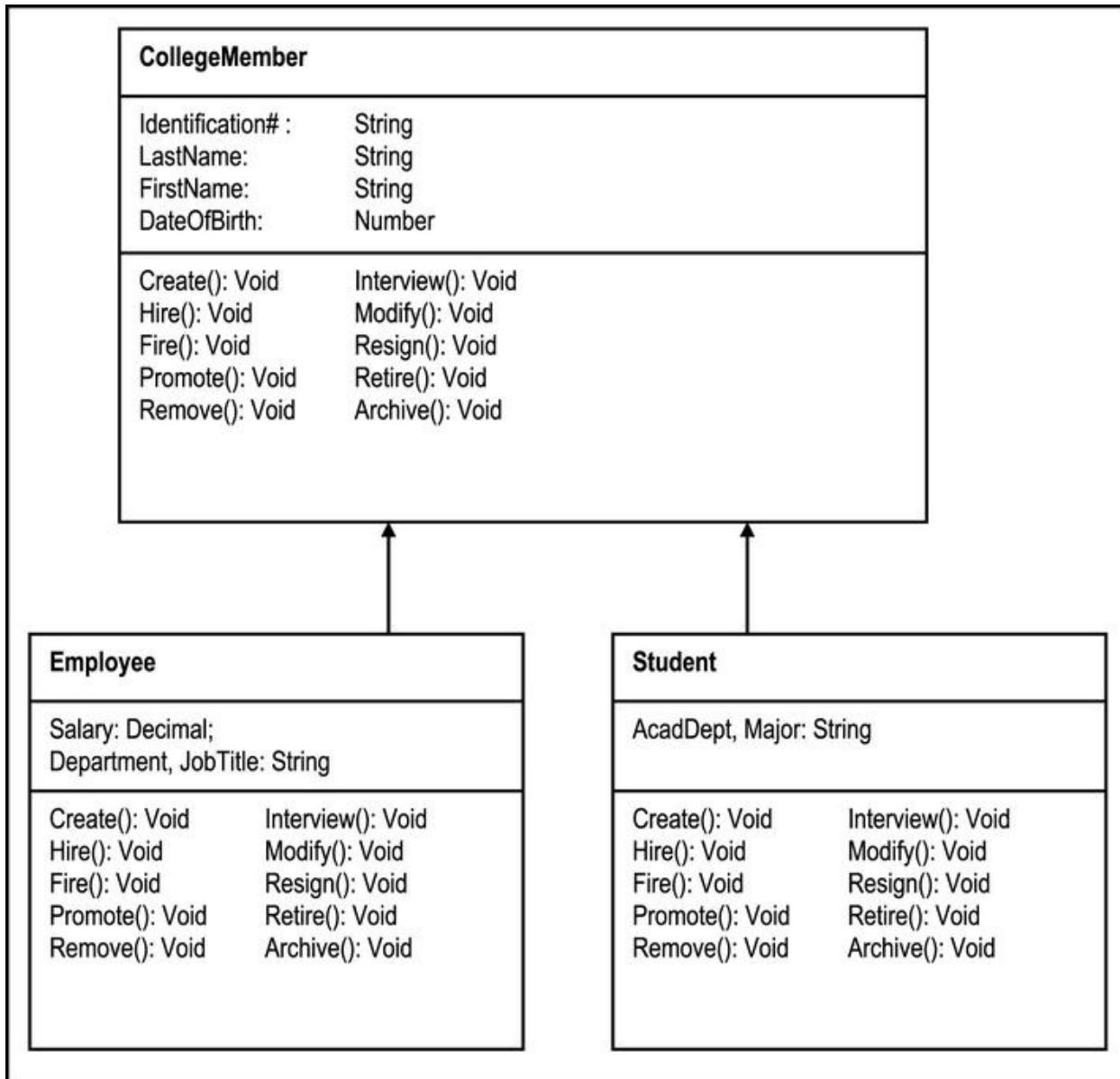


Figure A5-5. ORD Depicting Inheritance in a College Community

Moving to another example, [Figure A5-6](#) illustrates a configuration of five object types in what is called a *star schema*: A central object type (**SalesSummary**) is surrounded by a set of object types (in this case **Location**, **TimePeriod**, **Product**, and **ProductLine**). Each object type forms a 1:M relationship with the central object type. The star schema represented in the figure relates to tracking sales by a marketing firm based on dimensions such as time, location, product line, and product. Star schemas are widely used in data modeling. However, a full discussion of this topic is not necessary for this course. For more information on the matter, see the recommended readings

([Foster, 2010] and [Hoffer, 2007]).

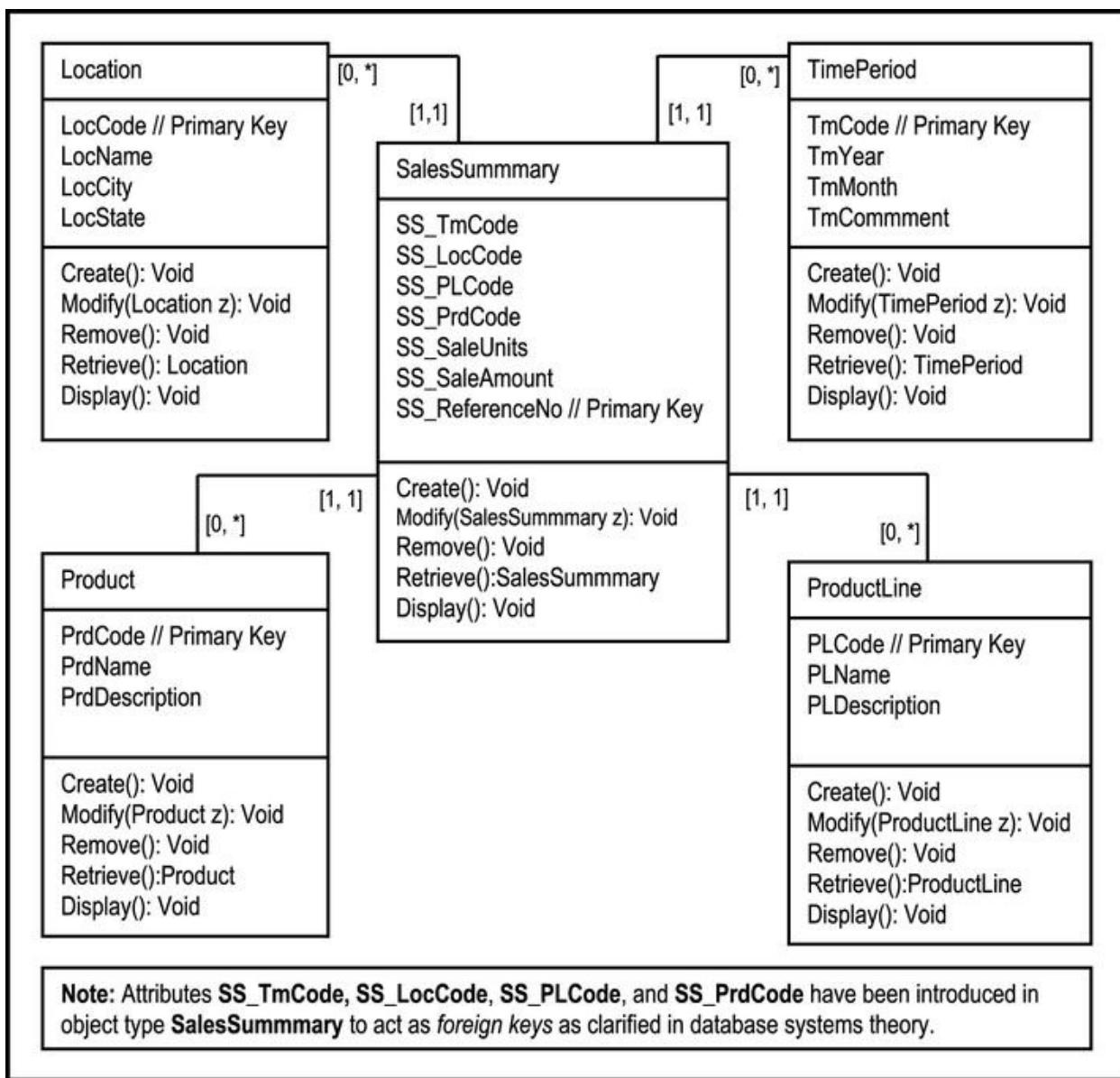
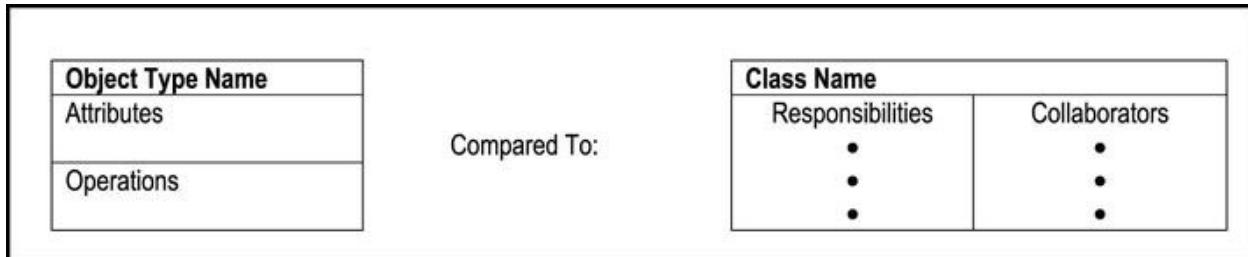


Figure A5-6. ORD for Tracking Sales Summary for a Large Marketing Company

Note Since ORD can grow bulky rather quickly, it is common practice to deemphasize (or even sometimes omit) the details relating to attributes and operations on the ORD. Some tools provide a plus sign (+) to expand a related section of the diagram, or a minus sign (-) to contract a related section.

A5.5.2 CRC Card

The CRC card can also be very useful in providing details about a class (which is the implementation of an object type). For the purpose of comparison, [Figure A5-7](#) summarizes the information contained in an OSD as well as a CRC card.



[Figure A5-7](#). Object Structure Diagram versus the CRC Card

Traditionally, CRC cards were used manually to assist in the analysis of the software system. Designers would literally prepare a deck of CRC cards for the object types comprising the system (one CRC card per object type), and use them during brainstorming sessions to assist in refining and finalizing the structure and role of each object type of the system (review section A4.1.9). To bring this technique to a contemporary scenario, the CRC card can be easily stored electronically, and used in not only refining but also modeling the software system.

A5.6 Avoiding Multiple Inheritance Relationships

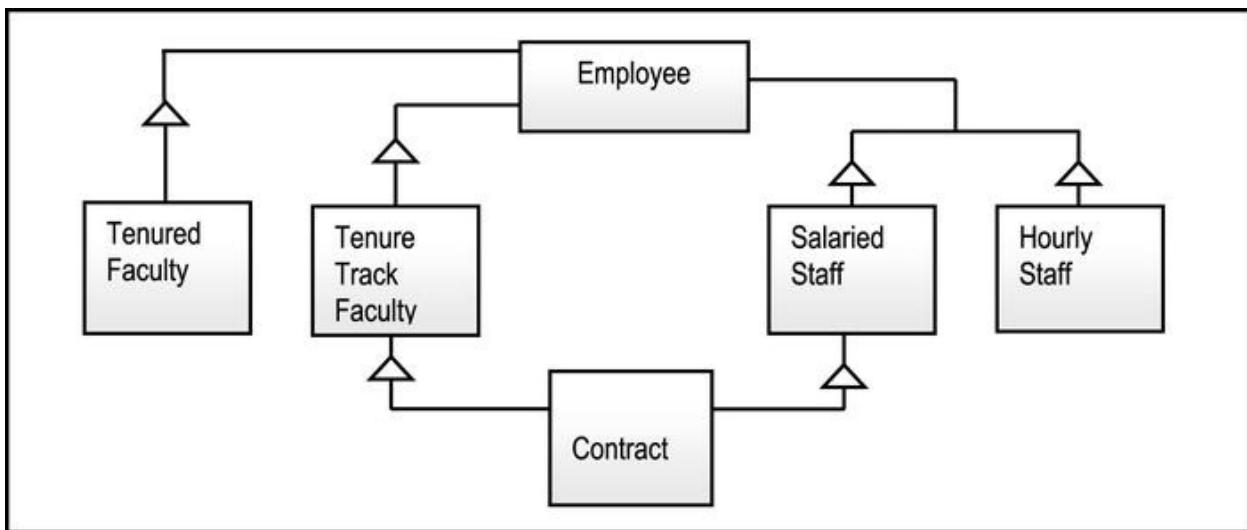
Dealing with multiple inheritances can be a challenge. You will recall from your OO programming, that they can cause confusion. Because of this, some OO programming languages (OOPLs) do not support them. James Rumbaugh in [Rumbaugh, 1991] describes three techniques for circumventing multiple inheritances; they are paraphrased here. The techniques (called *workarounds*) allow for avoidance of multiple inheritances in one of three ways:

- Delegation using aggregation
- Delegation and inheritance

- Nested generalization

A fourth approach for circumventing multiple inheritances is the use of interfaces as described in [appendix 2](#). This approach is supported quite nicely in the Java programming language.

[Figure A5-8](#) illustrates a multiple inheritance problem to be addressed. Let us examine how this can be resolved using the above-mentioned approaches, and as described in [Rumbaugh, 1991]. The first thing to note is that based on the figure, any alternate configuration should facilitate at least five categories of employees: tenured faculty, tenure track faculty, tenure track contractor, salaried contractor (no tenure track), and hourly paid staff. Now let us examine Rumbaugh's workarounds.



[Figure A5-8](#). A Multiple Inheritance Problem

A5.6.1 Delegation Using Aggregation

The delegation via aggregation technique involves the introduction of abstract object types that are composed of other types. [Figure A5-9](#) illustrates a solution to the multiple inheritance problem of [Figure A5-8](#), using aggregation. Notice the splitting of **Salaried Staff** into two separate object types, namely **Staff** and **Salaried**. This is necessary since there could be salaried or hourly-paid staff members. The abstract object types introduced are **Faculty** and **Staff**. A quick visual examination will also reveal that the minimum five categories of employees are facilitated in the figure.

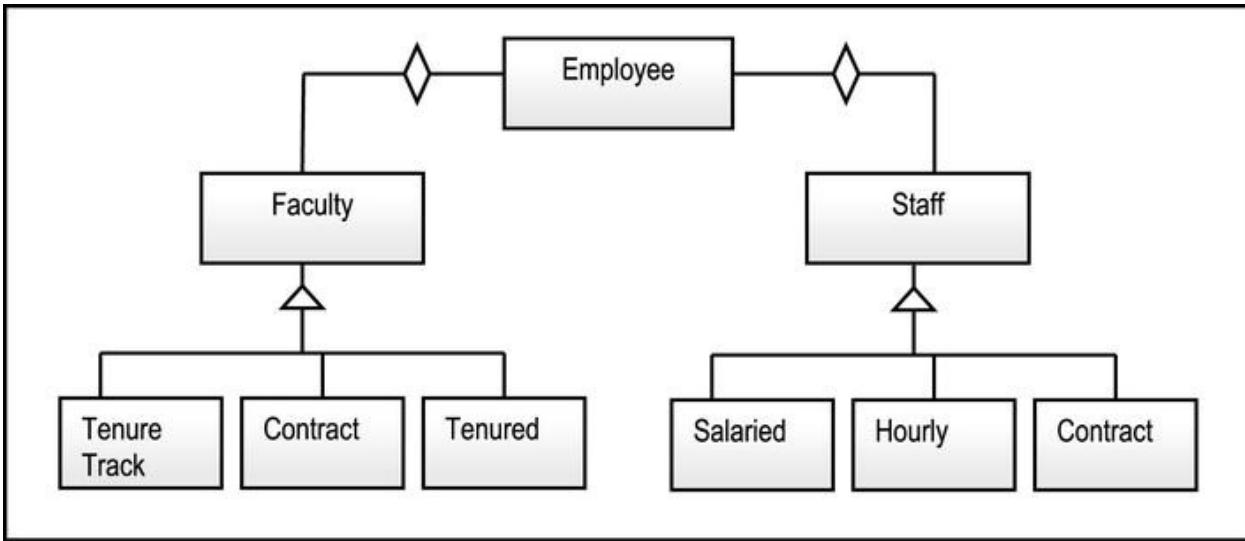


Figure A5-9. Multiple Inheritances using Delegation

Note The introduced abstract object type needn't have actual data attributes, and may merely consist of abstract operations (methods), which are overridden in the respective subtypes. This is particularly advantageous if you are using a purely object-oriented implementation language such as Java. This is the preferred approach for dealing with multiple inheritances.

A5.6.2 Delegation and Inheritance

In the delegation and inheritance technique, we inherit the most important class and delegate the rest. [Figure A5-10](#) illustrates a solution to the multiple inheritance problem of [Figure A5-8](#), using this approach. The original problem did not indicate which class is the most important, a judgment call was made to inherit on the faculty side, and delegate on the staff side. With this approach, you must be prepared to make such judgments. The role of the abstract class **Staff** is identical to the explanation in the previous subsection. Also note that the minimum five categories of employees are again facilitated.

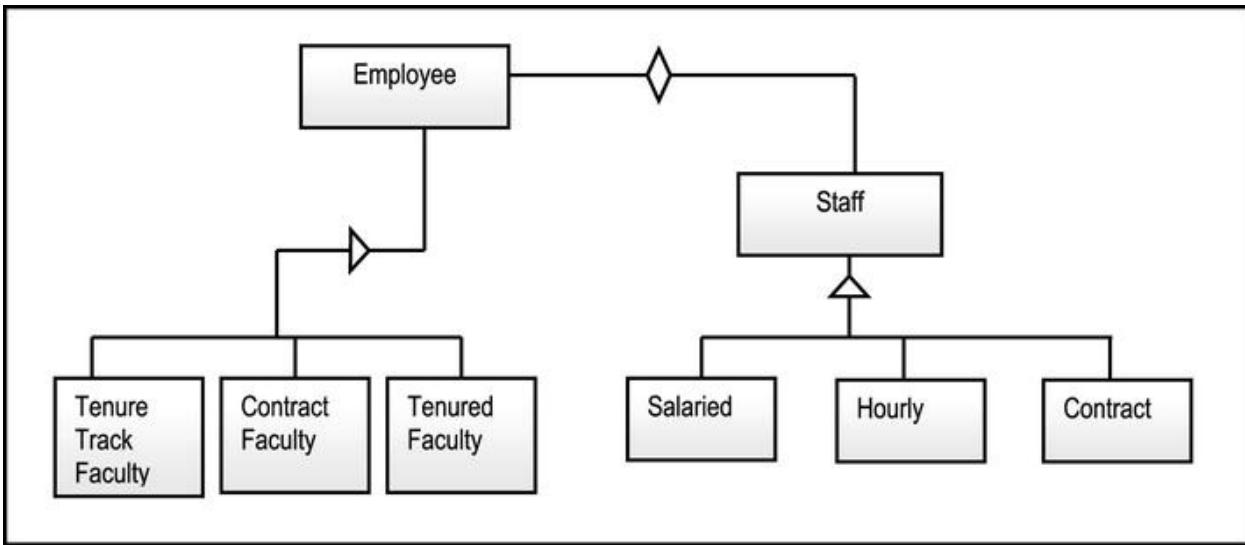


Figure A5-10. Multiple Inheritance via Inheritance and Delegation

A5.6.3 Nested generalization

The nested generalizations technique involves factoring one generalization first, then the other, until all possibilities are covered. It involves the introduction of abstract classes where necessary, in order to facilitate useful generalizations.

[Figure A5-11](#) illustrates a solution to the multiple inheritance problem of [Figure A5-8](#), using nested generalization. Notice the introduction of three abstract object types, namely **Faculty**, **Contract**, and **Staff**. Also observe that as in the two previous approaches, the minimum five categories of employees are facilitated.

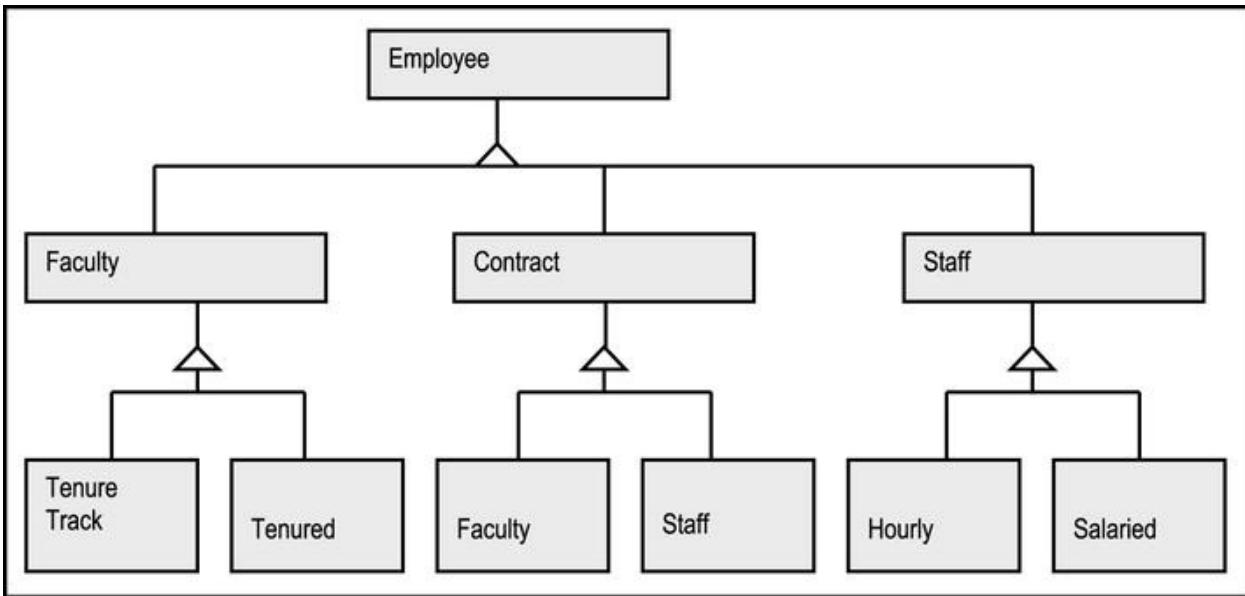


Figure A5-11. Multiple Inheritance via Nested Generalization

While this approach is very straightforward, it is not always recommended, since it often violates the principle of Occam's Razor by significantly increasing the number of classes to be managed.

A5.7 Top-Down versus Bottom-Up

You may conduct your object categorization by using either a top-down approach, or a bottom-up approach. In practice, it is a good habit to use both approaches — one as a check-and-balance mechanism to the other. Invariably, your implementation will be bottom-up (you have to create classes before you can use them).

A5.7.1 Top-Down Approach

For the top-down approach, use the following guidelines:

1. Start by looking at a summarized picture: determine what are the main facets of information are to be managed.
2. Break down the facets into constituents and sub-constituents as necessary (avoiding unnecessary

indentation levels).

3. Consider the facets as system modules or sub-systems, depending on the size of your project. Then consider the constituents and sub-constituents as object types.
4. A final step — not required for your ITC, but required for your database specification — is to identify and define for each object type, a set of properties (data attributes and allowable operations).

A5.7.2 Bottom-Up Approach

For the bottom-up approach, use the following guidelines:

1. Start out by identifying object types (tangible as well as intangible ones).
2. Identify and define for each object type, a set of properties (data attributes and allowable operations).
3. For each object type, provide an appropriate descriptive name.
4. Organize related object types into logical groups. These groups will constitute your super-types, system modules and/or subsystems (depending on the complexity of the project).
5. Integrate all modules and/or subsystems into one integrated system.

A5.8 Summary and Concluding Remarks

Here is a summary of what has been discussed in this chapter:

- The first step in object categorization is to identify the

relationships that exist among object types. The different types of relationships are 1:1, 1:M, M:1, M:M, subtype-super-type, aggregation, and component relationships.

- Once the relationships have been identified, they should be incorporated in the model of the software system. This can be done via diagramming techniques such as fern diagrams, ORDs, and ITCs.
- Details about object types can be modeled via techniques such as OSDs, CRC cards, and O/ESGs.
- Depending on the software development tool at your disposal, it may be advisable to avoid multiple inheritances in your design, since they could pose problems during software development. This can be done via any of four strategies: delegation using aggregation, delegation and inheritance, nested generalizations, and interfaces.
- It is a good habit to employ both top-down and bottom-up approaches to balance each other, as you design the software system.

Having addressed the structure and interrelationships of object types making up the software system, our next task is to design the behavior of objects in the system. The next chapter will address this.

A5.9 References and/or Recommended Reading

[Foster, 2010] Foster, Elvis C. with Shripad V. Godbole. *Database Systems: A Pragmatic Approach*. Bloomington, IN: Xlibris Publishing, 2010. See [chapters 3 – 5](#).

[Hoffer, 2007] Hoffer, Jeffrey A., Mary B. Prescott and Fred R. McFadden. *Modern Database Management* 8th ed. Upper Saddle River, NJ: Prentice Hall, 2007. See [chapter 11](#).

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development With UML and Java*. Upper Saddle River, NJ: Prentice Hall, 2002. See [chapter 8](#).

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1993. See [chapters 6 and 7](#).

[Rumbaugh, 1991] Rumbaugh, James, et. al. *Object Oriented Modeling And Design*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1991. See [chapter 4](#).

APPENDIX 6



Specifying Object Behavior

This chapter focuses on specifying object behavior. The chapter has the following captions:

- Use-cases
- States and State Transitions
- Finite State Machines
- Event Diagrams
- Triggers
- Activity Diagrams
- Sequence Diagrams and Collaboration Diagrams
- Object Flow Diagrams
- Summary and Concluding Remarks

A6.1 Use-cases

The technique called *use-case* was first introduced by Ivar Jacobson in [Jacobson, 1992], and has since then been embraced by the software engineering industry. A use-case is a representation of all possible interactions (via messages) among a system and one or more *actors* in response to some initial stimulus by one of its actors. The use-case describes the functionality provided by a system, in order to yield a visible result for one of its actors.

In constructing use-cases, a few terms need to be clarified:

Actor: An actor is an external entity that interfaces with the system. Actors could be individuals as well as other external systems that interact with the system in question.

Scenario: A *scenario* is a specific instance of a use-case. The use-case represents a set of scenarios that involve an actor's engagement.

Use-case Bundle: A use-case bundle refers to a group of related use-cases. Bundling may be based on the fact that the use-cases share the same actor and/or state, common entities, or common workflow.

The use-case allows the user to view the system from a high level, focusing on possible interactions between the system and its actors. In a way, use-cases should remind you of data flow diagrams (DFDs) in functional design.

A6.1.1 Symbols Used in Use-case

[Figure A6-1](#) includes the symbols used in use-case diagrams. As an example, consider the CUAIS project, and let us focus on the scenario where a student may register for a course, change his/her registration (for instance, auditing a course previously registered for), or drop a course. In the same vein, a department representative could add a course, cancel a course, or modify a course offering (changing its room and/or time period). [Figure A6-2](#) illustrates a use-case diagram for this scenario.

System

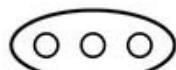
Shows the system as a black box which provides certain functionalities, but pays no concern to how these functionalities are provided.



The use-case. Shows how objects of the system map to business processes that end users need to perform.



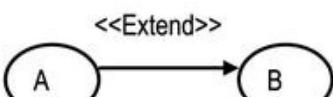
Actor that interacts with the system.



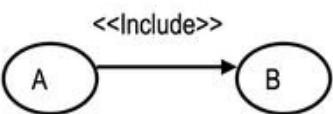
Shows component processes making up a business process.



Request from a requestor to a provider of a service.



Use-case A extends use-case B.



Use-case A includes use-case B.

Figure A6-1. Symbols Used in Use-case Diagrams

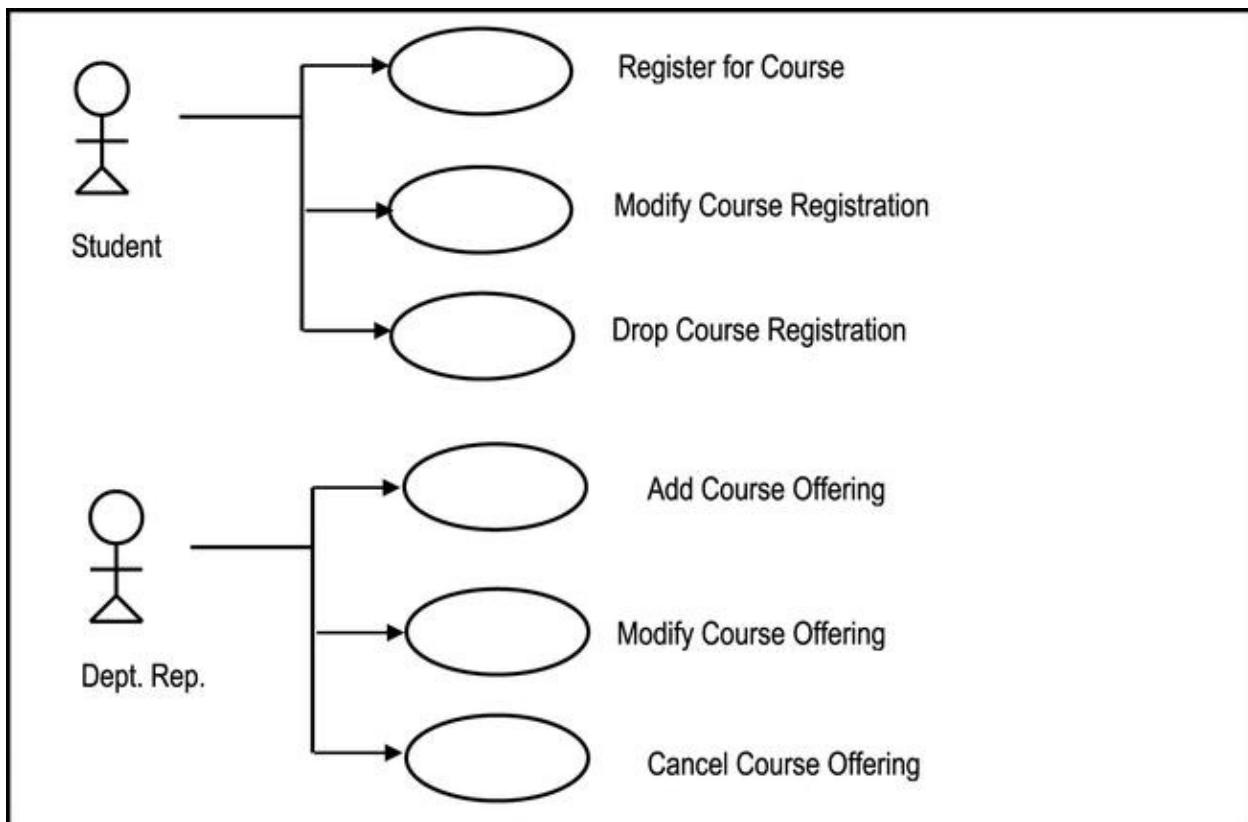


Figure A6-2. Use-case for Registration Process

A6.1.2 Types of Use-cases

Use-case diagrams can be constructed for different levels of system functionality (they could be *high-level* or *low-level*):

- A high-level operation provides the essence of the business values provided.
- A low-level operation provides more detail about component or concurrent activities and their order.

Use-cases may also be classified in terms of *primary* or *secondary* operations that they represent:

- Primary operations relate directly to the essential business functionalities of the system. They are the options provided to the end user.

- Secondary operations are those operations, which, though not directly accessible by the end user, are essential for the delivery of a coherent, robust system. They are also referred to as subservient operations.

A third perspective is to differentiate *essential operations* from *concrete operations*:

- Essential operations are business solutions that are platform independent.
- Concrete operations are design dependent.

A use-case may be high-level or low-level; primary or secondary; essential or concrete. It may also involve a mix of these classifications. Additionally, a use-case may *include* a previously defined use-case, or it may *extend* a previously defined use-case.

A6.1.3 Information Conveyed by a Use-case

A use-case may be used to convey the following information:

- **Actors:** These are the participating entities.
- **Relationships with Other Use-cases:** Two kinds of relationships are possible — the *include* relationship and the *extend* relationship.
- **Pre-conditions:** A pre-condition is a condition that must be satisfied prior to the invocation of the use-case.
- **Post-conditions:** A post-condition is a condition (or state) that must hold after the execution of the use-case.
- **Details:** This entails the step-by-step interactions among participating objects in the use-case.
- **Constraints:** Any constraint (apart from pre-conditions and post-conditions) that applies to the use-case must be specified. Constraints may be with respect to values and resources available for manipulation.

- **Exceptions:** This involves the identification of all possible errors that might occur in the use-case.
- **Variants:** This includes all variations that may apply.

Not all of this information may be relevant for each use-case; [Figure A6-3](#) provides some guidelines.

Information	HPE Use-Case	LPSE Use-Case	LPC Use-Case	LPSC Use-Case
Actors	✓	✓	✓	✓
Relationships		✓		✓
Pre-conditions	✓	✓	✓	✓
Post-conditions	✓	✓	✓	✓
Details	✓	✓	✓	✓
Constraints	✓	✓	✓	✓
Exceptions		✓		✓
Variants	✓	✓	✓	✓

Key:
HPE = High-level primary essential
LPSE = Low-level primary/secondary essential
LPC = Low-level primary concrete
LPSC = Low-level primary/secondary concrete

[Figure A6-3](#). Information Associated With Different Kinds of Use-cases

A6.1.4 Bundling Use-cases and Putting Them to Use

As pointed out earlier, a use-case bundle refers to a group of related use-case. Typically, the component use-cases make sense when the bundle is viewed from a business perspective. Use-cases may be bundled using any of the following criteria:

Same Actor, Same State: All use-cases involving the same actor and the same state are grouped. The use-cases of [Figure A6-2](#) meet this criterion.

Common Entities: Use-cases are bundled based on the fact that they use and/or impact the same entities. The rule-of-thumb method for object identification, discussed in [appendix 4](#) (section A4.1.10), flows naturally into this approach. For example, in designing an inventory management system, one may identify and refine use-cases relating to the purchase of an invoice. These would typically involve adding, modifying, deleting and inquiring on invoices.

Specific Workflow: Use-cases that relate to a particular actor (user) carrying out a functional responsibility are bundled together. More often than not, same-actor-same-state use-cases qualify as specific workflow use-cases. [Figure A6-2](#) therefore applies here also. The idea here is that if we can identify, define and implement all the operations which each actor of the system requires, we would achieve a comprehensive system meeting user requirements.

Once the use-cases have been identified and represented (via use-case diagrams), the next step is to use the information to refine your system model. Remember, your focus is object behavior. The rest of the chapter will concentrate on some methodologies for analyzing and modeling system behavior.

A6.2 States and State Transition

An object can exist in one of many states. A state life cycle is the sequence of states that the object may be in during its useful life.

State changes trigger operations; operations invoke methods, which further result in state changes. For example: An employee may be hired, promoted, and eventually fired. Each of these operations results in state changes which are recorded in the object's data.

A state transition diagram shows the allowable states of an object. [Figure A6-4](#) and [Figure A6-5](#) provide examples of state transition diagrams.

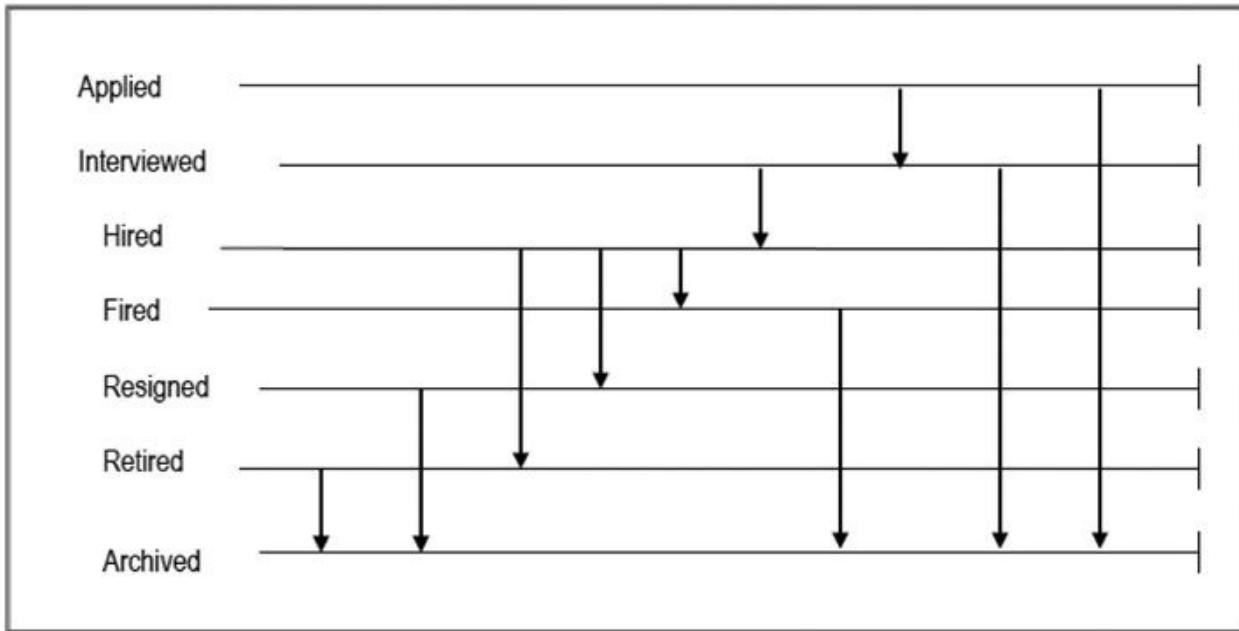


Figure A6-4. State Transition Diagram for Employee Object

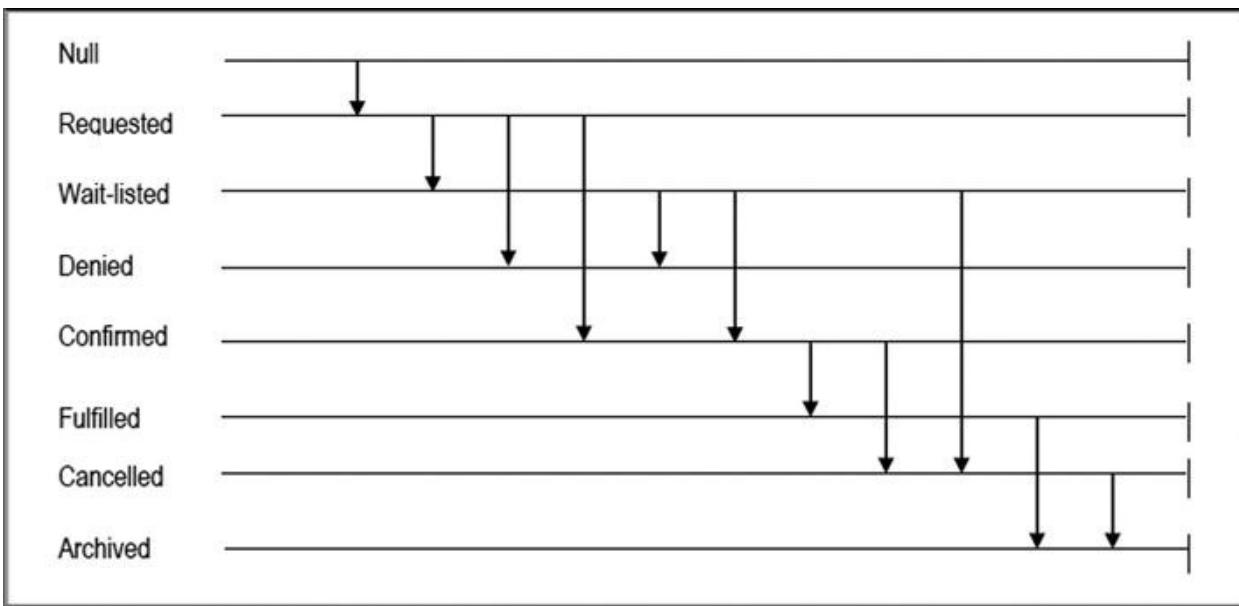


Figure A6-5. State Transition Diagram for Airline Booking Object

An object may have more than one set of states. For example, in addition to the states indicated in [Figure A6-5](#), a second set of states for a confirmed airline booking could be:

- Unpaid
- Deposit Paid

- Fully Paid
- Needs Refund
- Refunded

These additional states would exist between the states **Confirmed** and **Fulfilled**.

An object may have several states, which in turn may have sub-states. (In some OO-CASE tools, the state which has sub-states is usually indicated by a ‘+’. Clicking here causes the sub-states to be displayed.)

The two sets of states may be merged in the interest of simplicity. Alternately, the two sets of states may be linked by some conditionality (rule). For example: “Booking cannot be confirmed until payment state is either Deposit Paid or Fully Paid.”

A6.3 Finite State Machines

An alternative to the state transition diagram and the event diagrams is the finite state machine (FSM). This technique, though existent before object technology, finds very useful application in OOSE. A class may be considered as an FSM with methods, inheritance and encapsulation.

In an FSM (also referred to as a *state-diagram*) nodes are states; arcs are transitions labeled by operation names (the label on a transition arc is the operation name causing the transition). State name is written inside the node. Typically, an FSM is developed for a class; the operations represent the set of operations defined on the class; the state transitions represent the set of possible transitions for any instance of the class. [Figure A6-6](#) illustrates an FSM for a class called **Employee**. This figure includes all the information provided in [Figure A6-4](#), but also provides additional information.

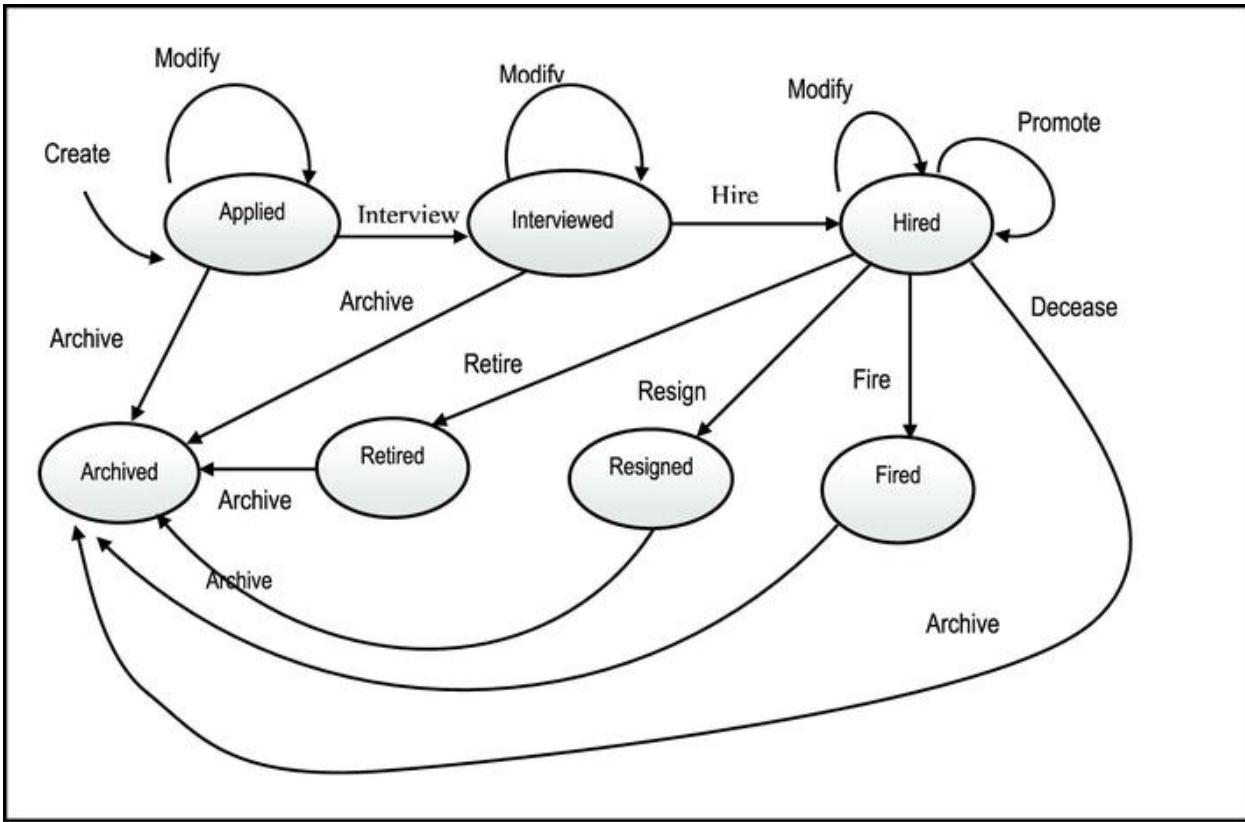


Figure A6-6. FSM for Employee Object

On the FSM, there is no distinction between an *event* (discussed next) and an operation. This is reasonable, since as you will soon see, events and operations are intrinsically related (one leads to the other).

One significant advantage of the FSM is that you can show operations that do not cause change of state. A review of [Figure A5-6](#) reveals that in several cases, there is a **Modify** operation that does not necessarily result in a state change. Another advantage is that the FSM is easy to draw, maintain, and understand.

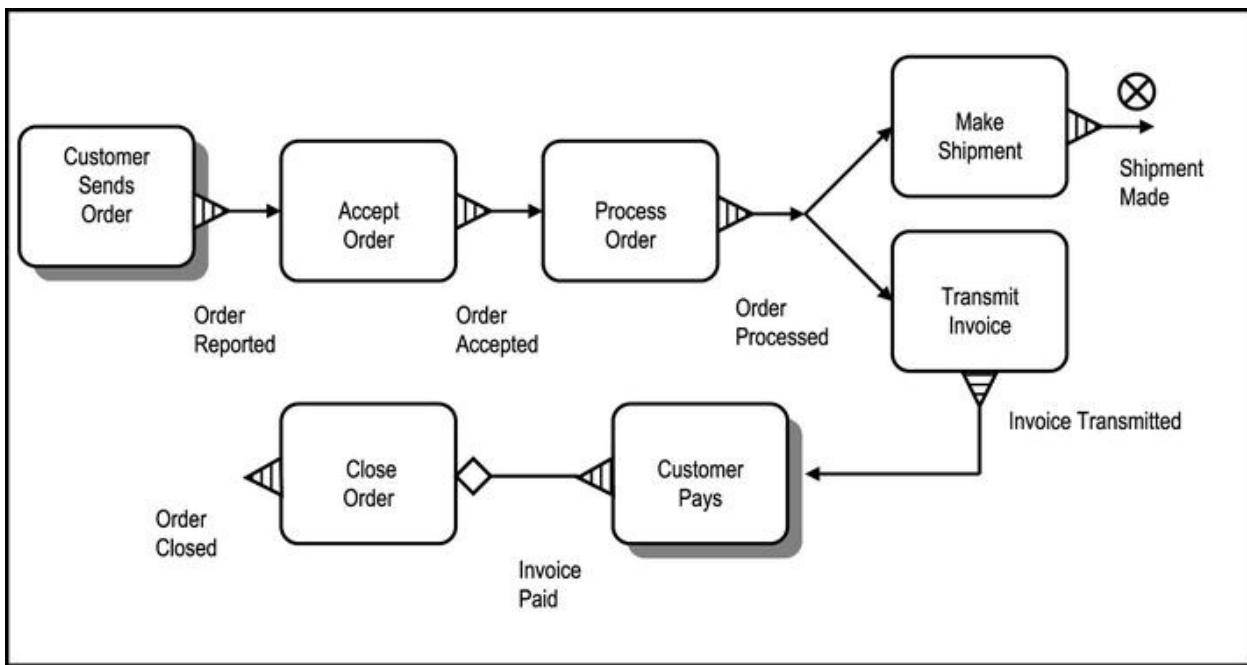
A6.4 Event Diagrams

[Martin, 1993] describes event diagrams as a means of modeling the interrelationship between operations and events, as summarized in this section.

An *event* is a noteworthy change in the state of an object. Events trigger operations. An operation may change the state of an object thereby causing another event. The event diagram shows these events and their associated

operations.

An event is represented by a filled in triangle; an operation is represented by a round-cornered box (shadowed if operation is external). When an event results from an operation, the triangle is attached to the operation box. An example is shown in [Figure A6-7](#), where an event diagram is shown for the processing of a purchase order.



[Figure A6-7](#). An Event Diagram for Processing a Purchase Order

The event diagram relates to the state transition diagram in the following way: the latter depicts state changes; the former depicts causes for those state changes. On the state transition diagram, each transition may be elaborated by an event diagram. State transition diagram and event diagram are complementary. The one assists in ensuring the accuracy of the other. With an OO-CASE tool, the designer chooses to work with either.

The operation is isolated from cause and effect. Each object type (class) has an associated state transition diagram and event diagram(s).

The event diagram is to OOSE what the program flowchart is to the more traditional (function-oriented) approach to software engineering. In more complex environments, operation specifications may be required to compliment the event diagrams.

An operation may have pre-condition(s) and post-condition(s). Pre-conditions must be true before the operation can take place; post-conditions must

hold after the operation is completed. Pre-conditions and post-conditions are referred to as control conditions. The diamond symbol of [Figure A6-6](#) indicates that pre-condition(s) exist for the operation **Close Order**. With an OO-CASE, pre-conditions can be clicked for details.

Just as there are object types, there are event types. The analyst is more interested in event types than the number of occurrences of each event. Typically event types describe the following kinds of changes:

- Creation of new object
- Termination of object
- An object changes classification
- An object's attribute(s) is (are) changed
- Object is classified as an instance of another type; for example: **Lecturer** becomes **Department Chair**
- Object is declassified; for example: **Lecturer** ceases to be **Department Chair**

Control conditions are particularly useful when multiple trigger rules lead to a specific operation, as illustrated in [Figure A6-8](#). In the figure, **Operation A**, **Operation B**, and **Operation C** must occur before **Operation D** can occur.

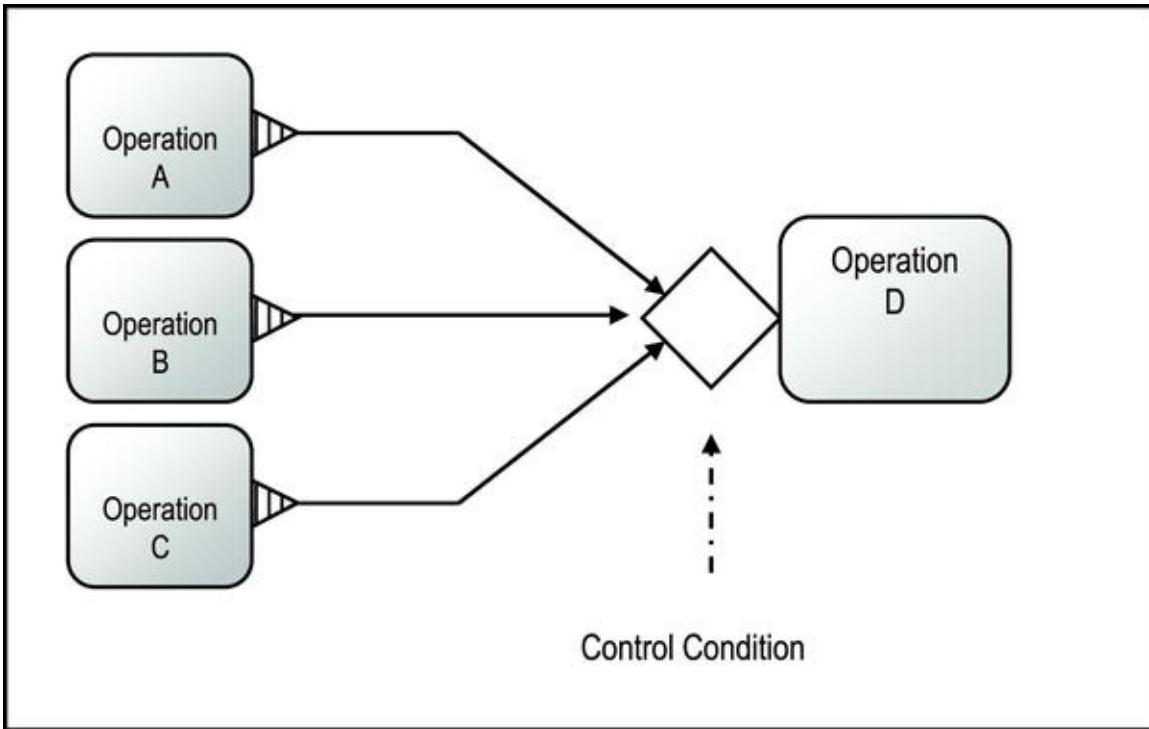
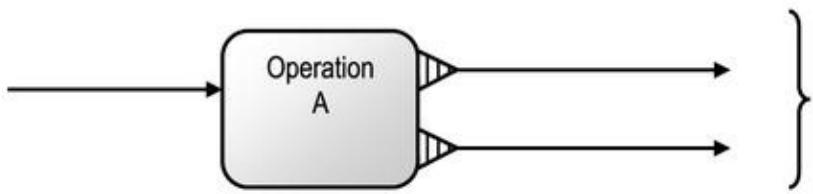


Figure A6-8. Illustrating Control Condition

Events may also have subtypes and super-types as illustrated in [Figure A6-9](#). In this figure, **Operation A** has two sub-events.



Since the two events from operation A are mutually exclusive, it is better to show one event type as follows:

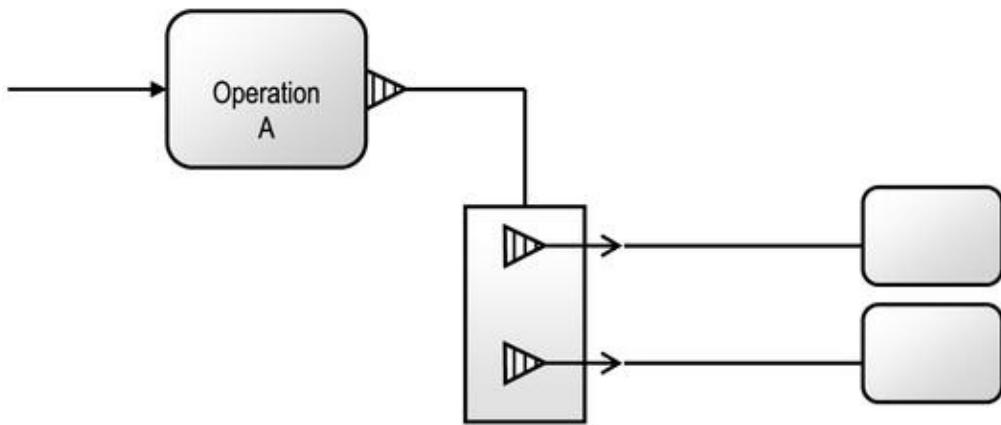


Figure A6-9. Illustrating Event Sub-types

A6.5 Triggers

[Martin, 1993] also describes and trigger rules — two related concepts that have become commonplace in contemporary software engineering environments. The line going from an event to the operation(s) it triggers, represents a trigger rule. The trigger rule defines the causal relationship between event and operation(s). An event may trigger one or more operations; also, an operation may be caused by multiple triggers ([Figure A6-10](#) illustrates).

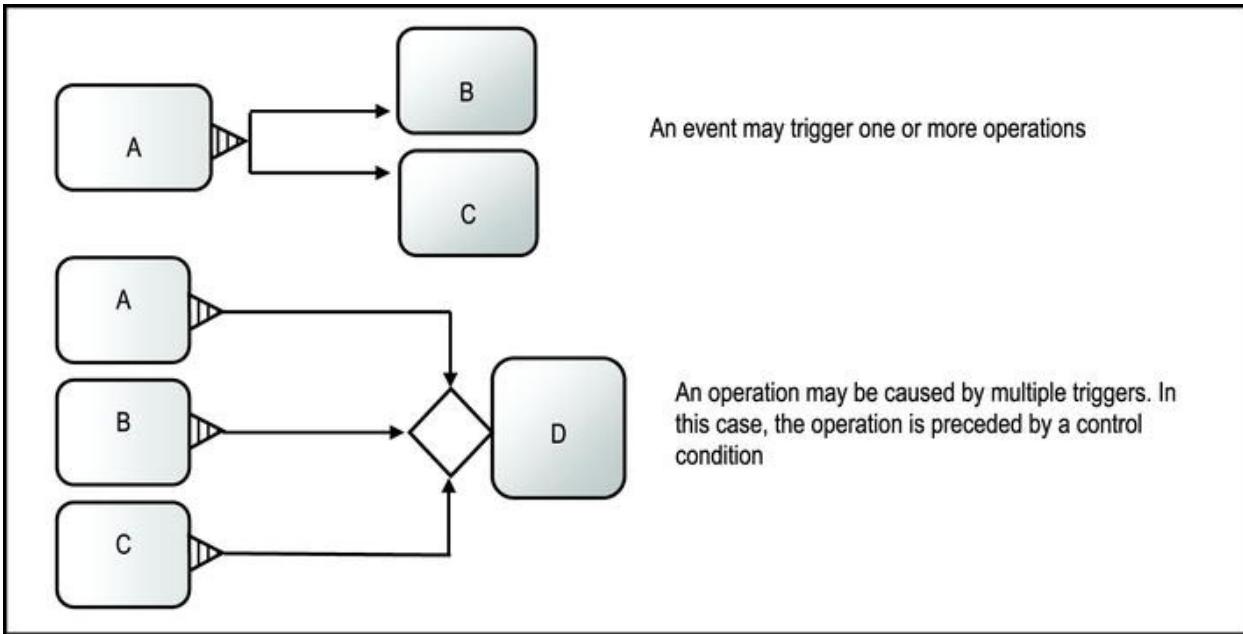


Figure A6-10. Illustrating the Relationship between Triggers and Operations

Like objects types, operations may have subtypes and super-types as illustrated in [Figure A6-11](#).

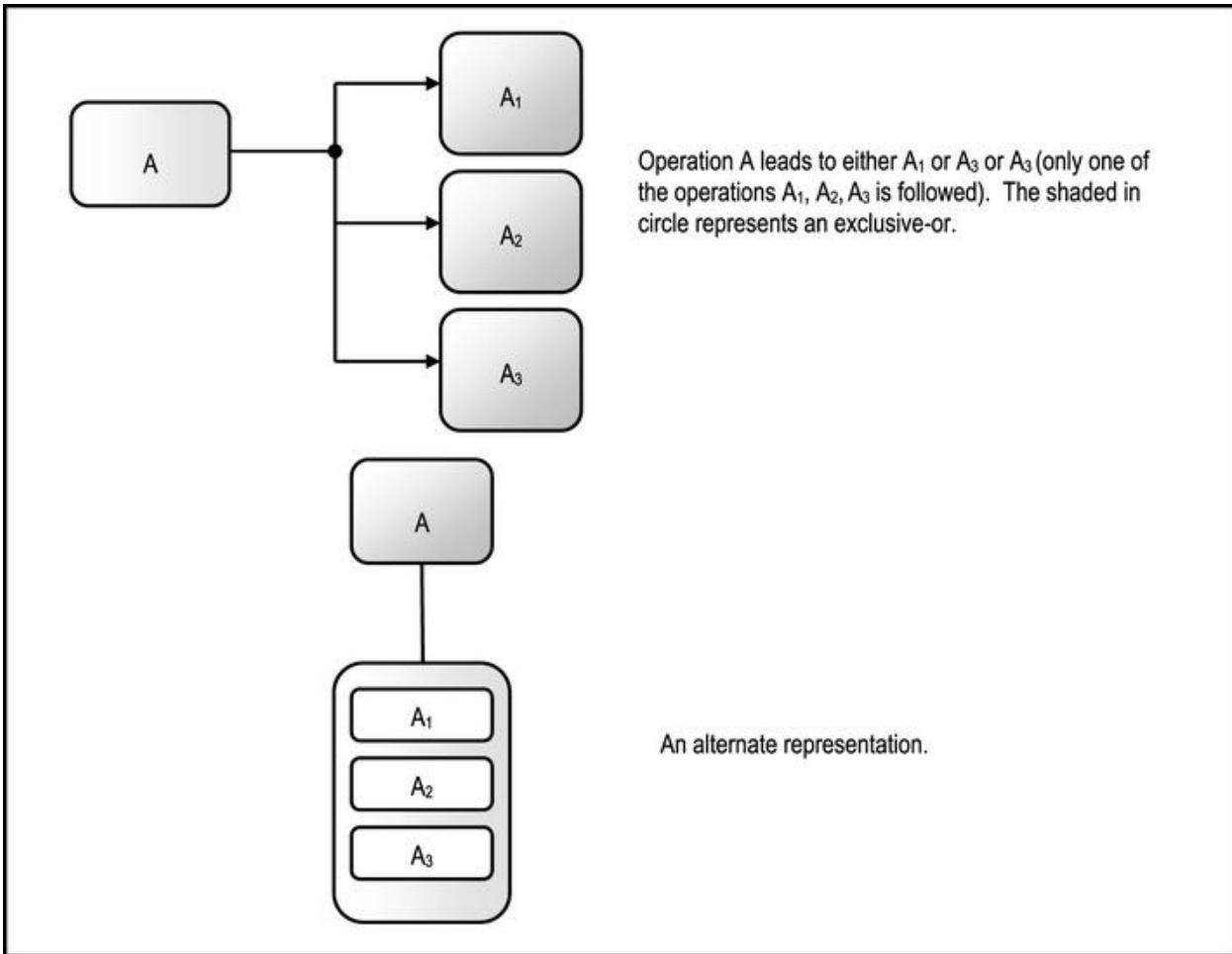


Figure A6-11. Illustrating Operation Sub-types

A6.6 Activity Diagrams

As an alternative to event diagrams, UML supports *activity diagrams*. An activity diagram describes how related activities (computations and workflow) are coordinated. Bear in mind that activities are not existence independent, but are typically subservient to operations and other activities.

The activity diagram (graph) depicts activity states and the transition among activities in a workflow. An activity state represents the execution of a statement in a procedure, or the performance of an activity in a workflow.

The activity diagram conveys information about sequential activities as well as concurrent activities (threads). [Figure A6-12](#) shows the UML notations for activity diagrams, while [Figure A6-13](#) provides an example.

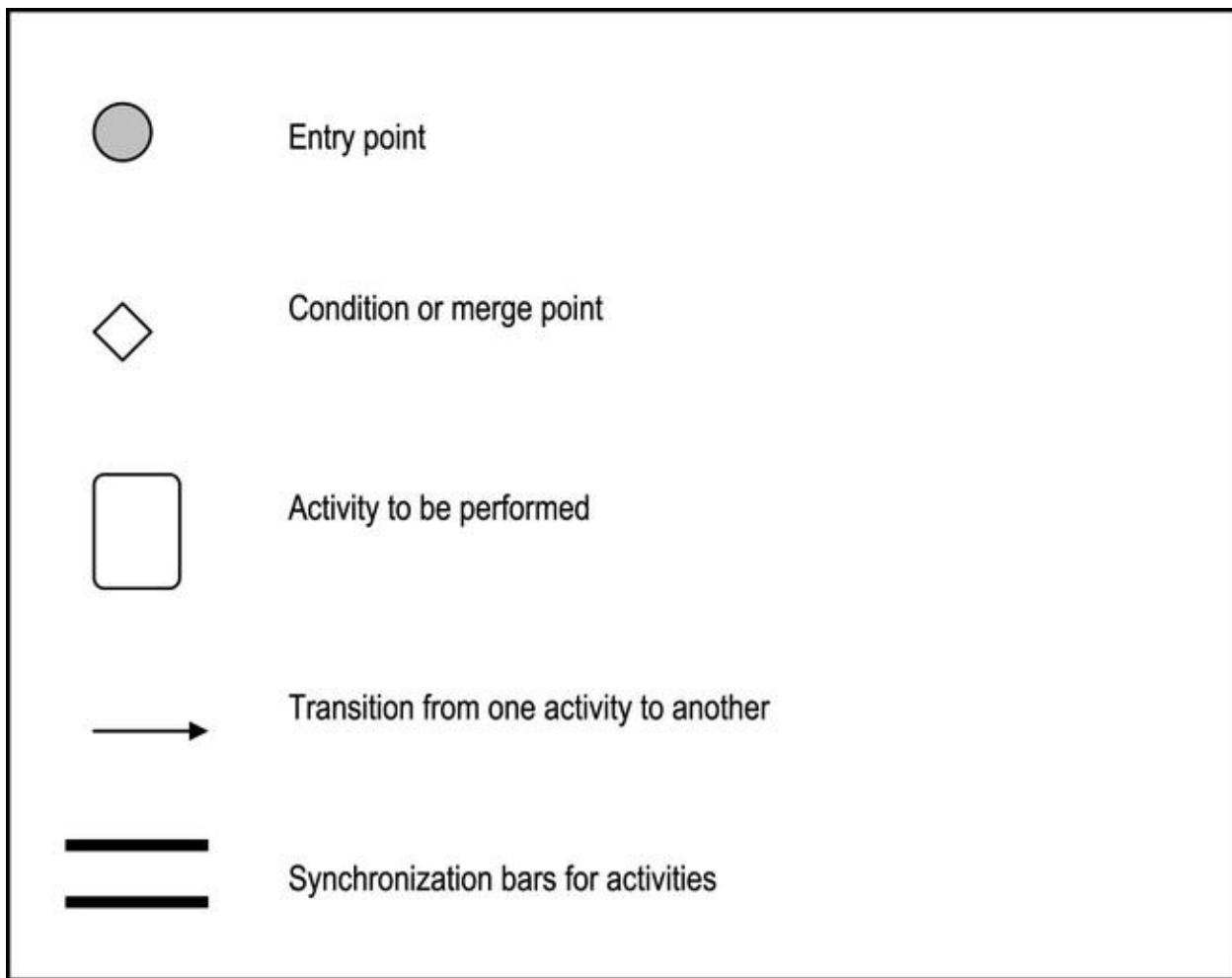


Figure A6-12. UML Notations for Activity Diagram

The activity diagram may contain *swim-lanes*: Whenever the situation arises where it is desirable to group related activities of a process into distinct partitions (these partitions may represent functional areas in a business), vertical or horizontal lines are used to define the partitions. The partitions are called swim-lanes. Swim-lanes could have been introduced for the three concurrent threads in [Figure A6-13](#) (they were omitted since their introduction would have cluttered the diagram).

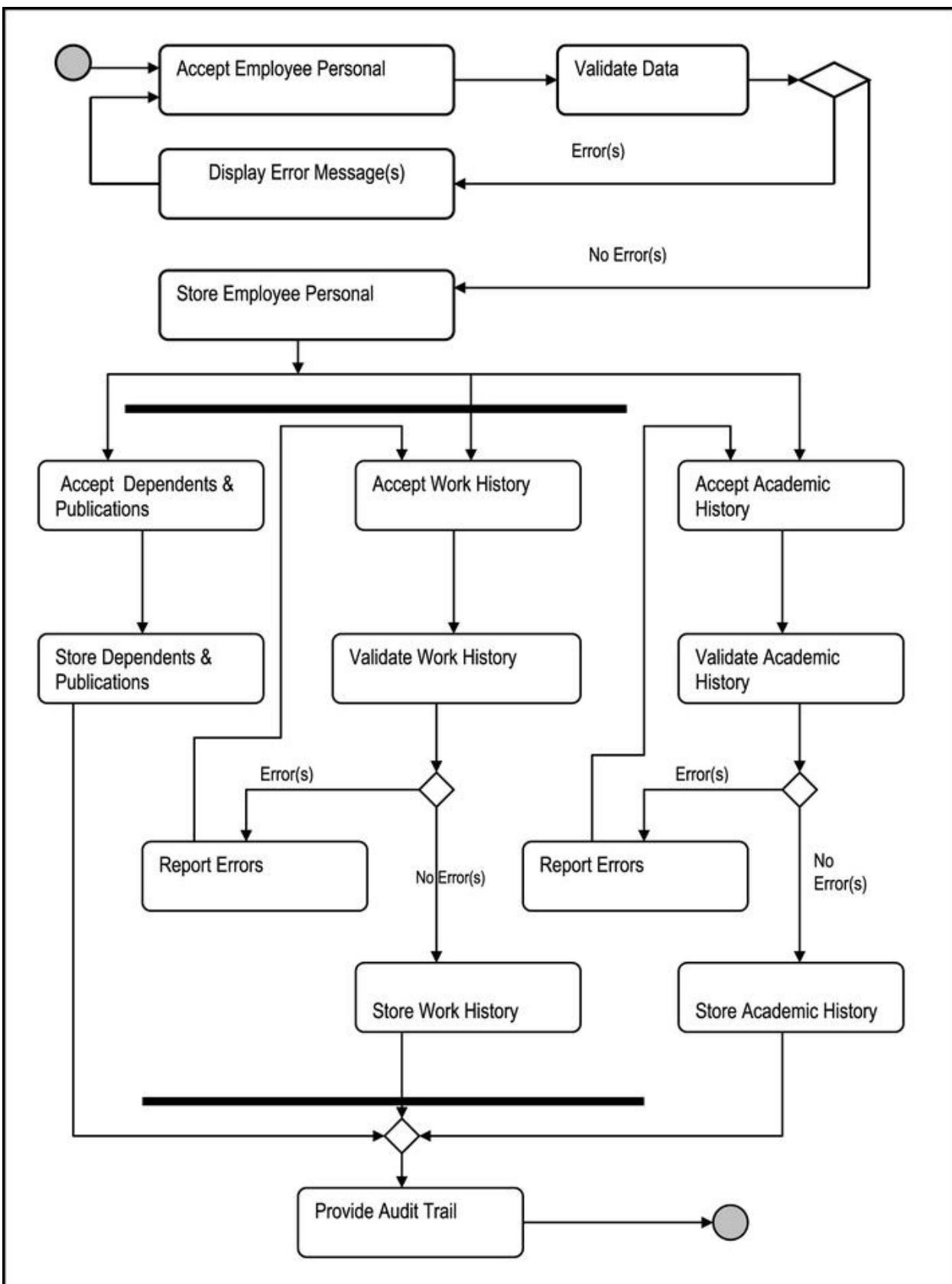


Figure A6-13. Activity Diagram for Adding Employee Information

A6.7 Sequence Diagrams and Collaboration Diagrams

Another UML notation is the *collaboration diagram*. A collaboration diagram (also described in older literature on object technology as an *object interaction diagram*) is a special class diagram that is used to depict various interactions among objects that participate in a process.

The collaboration diagram is useful in illustrating the implementation of a class operation, or the set of operations of a class. It includes permanent objects as well as objects created and/or destroyed during the process. **Figure A6-14** illustrates a collaboration diagram for an inquiry on student information in the CUAIS project of earlier discussions. A message from one object to another is represented as a directed arrow, labeled with the name of the service requested and an argument list.

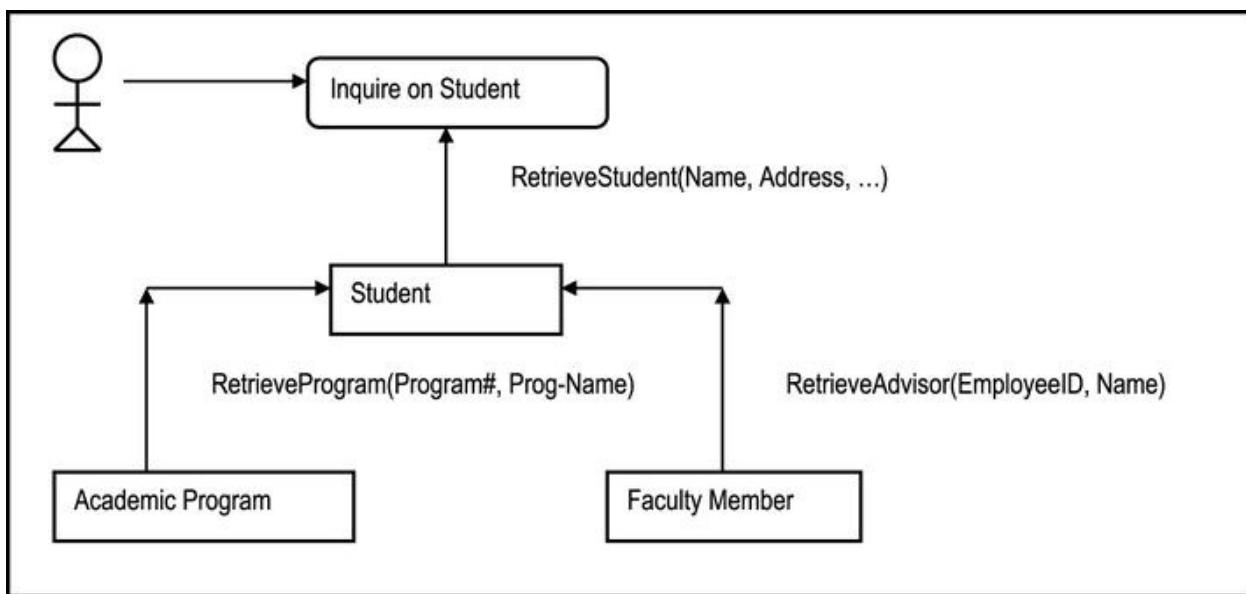


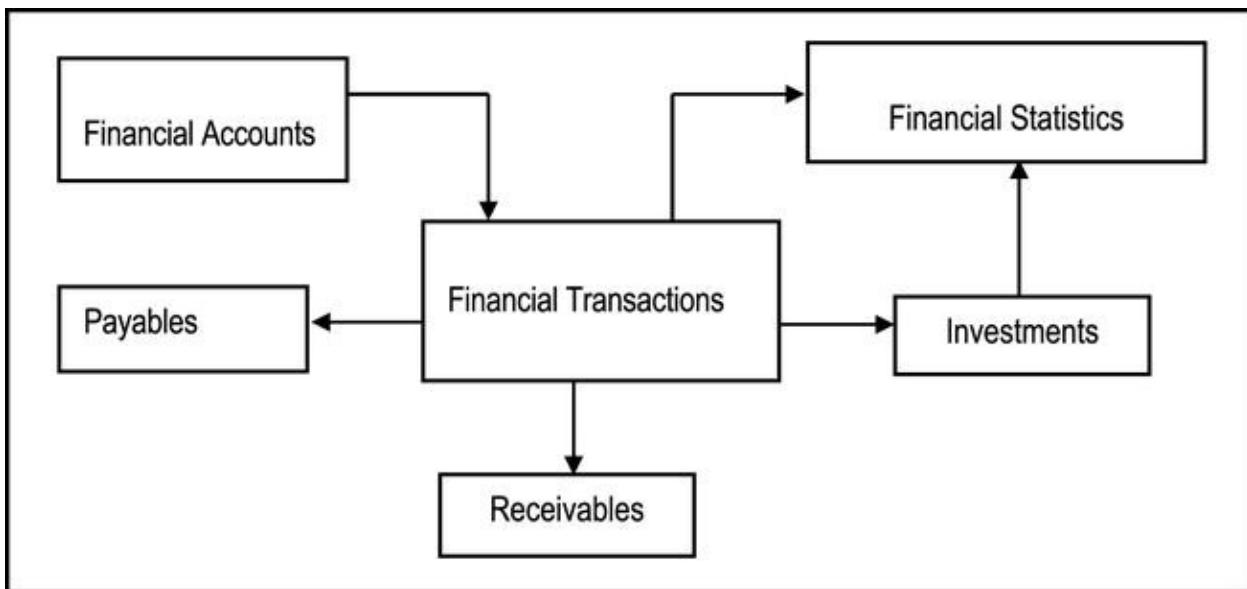
Figure A6-14. Collaboration Diagram for Inquiry on Student Information

A *sequence diagram* — another UML notation — shows how objects interact with each other (via messages) with respect to time, in a two dimensional chart. The vertical dimension indicates time, which increases down

the axis. The horizontal dimension indicates the objects in the interaction. Except for real time systems, sequence diagrams are not as widely used as the other techniques discussed; they are therefore not discussed any further. For additional information, see the recommended readings [Lee, 2002] and [Rumbaugh, 1999].

A6.8 Object Flow Diagrams

An *object flow diagram* (OFD) is similar to a high-level data flow diagram (DFD) in traditional software engineering. It is non-executable, and often used as a system overview diagram. It indicates communication lines among the main components (subsystems or super-classes) at a very high level. [Figure A6-15](#) illustrates an OFD for a financial management system, which assumes a client-server approach or a component-based approach for the system architecture. [Figure A6-16](#) illustrates another OFD — this time for the CUAIS project — which assumes a repository approach for the system architecture.



[Figure A6-15](#). An OFD for a Financial Management System

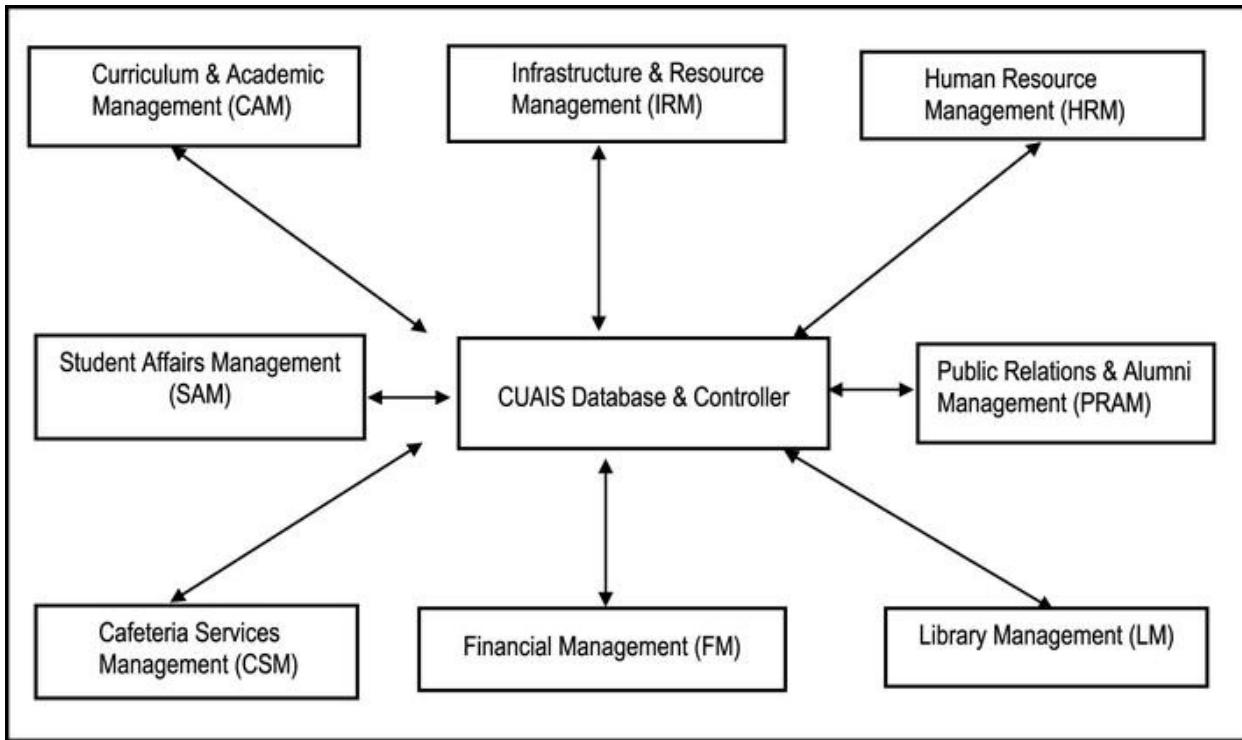


Figure A6-16. OFD for the CUAIS Project, Assuming Repository Model

A6.9 Summary and Concluding Remarks

Let us summarized what has been discussed in this chapter:

- A use-case is a representation of all possible interactions (via messages) among a system and one or more actors in response to some initial stimulus by one of its actors. The use-case describes the functionality provided by a system, in order to yield a visible result for one of its actors.
- A state transition diagram (STD) is a diagram that represents the possible states that an object can be in. An alternate diagram is a finite state machine (FSM). In addition to indicating the possible states, the FSM also shows the operations that lead to state changes.
- An event diagram represents the details of operations

comprising a system and the events they lead to.

- As an alternative to event diagrams, UML supports activity diagrams. An activity diagram describes how related activities (computations and workflow) are coordinated. The activity diagram depicts activity states and the transition among activities in a workflow. An activity state represents the execution of a statement in a procedure, or the performance of an activity in a workflow.
- A collaboration diagram is useful in illustrating the implementation of a class operation, or the set of operations of a class. It includes permanent objects as well as objects created and/or destroyed during the process.
- An object flow diagram (OFD) is often used as a system overview diagram. It indicates the main objects and activities of the system at a very high level.

Other techniques have been proposed for depicting object behavior, which have not been covered in this chapter. The ones covered are considered essential for constructing a comprehensive model of the software system or information infrastructure that has your focus. The next chapter discusses tools that support OOM.

A6.10 References and/or Recommended Reading

[Bruegge, 2000] Bruegge, Bernard and Allen H. Dutoit. *Object Oriented Software Engineering: Conquering Complex and Changing Systems*. Upper Saddle River, NJ: Prentice Hall, 2000. See [chapter 2](#).

[Jacobson, 1992] Jacobson, Ivar. *Object Oriented Software Engineering – A Use Case Approach*. Reading, MA: Addison-Wesley, 1992.

[Lee, 2002] Lee, Richard C. and William M. Tepfenhart. *Practical Object-Oriented Development WithUML and Java*. Upper Saddle River, NJ: Prentice Hall, 2002. See [chapters 3, 5, 6-8](#).

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1993. See [chapters 8](#) and [9](#).

[Rumbaugh, 1991] Rumbaugh, James, et. al. *Object Oriented Modeling And Design*. Eaglewood Cliffs, New Jersey: Prentice Hall, 1991. See [chapters 5](#) and [6](#).

[Rumbaugh, 1999] Rumbaugh, James, Ivar Jacobson and Grady Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999. See [chapter 8](#).

APPENDIX 7



Tools for Object-Oriented Methodologies

The previous six chapters discussed the fundamentals of OOM with respect to software design as well as information infrastructure design. This chapter discusses OOM tools. The chapter proceeds under the followings captions:

- Introduction
- Categories of CASE Tools
- Universal Database Management Systems
- Benefits of OO-CASE Tools and UDBMS Suites
- Object-Oriented Programming Languages
- Modeling and Code Generation
- Standards for OOM
- Summary and Concluding Remarks

A7.1 Introduction

Object technology and CASE technology fit naturally together; they are like “a match made in heaven”. The reason for this is twofold:

- Many intricacies of the OO approach (for example information hiding) are implemented by CASE tools at a level beyond user intervention and in a manner that is

transparent to the user (developer as well as end users).

- CASE enhances the OO approach and makes it more attractive. In fact, without CASE, the OO approach would have probably remained an alternative widely discussed in academic circles, but seldom used in practice. The techniques would simply be too arduous and time consuming to be worth the effort.

Early emergence of object technology relied on, and heavily emphasized OO programming. The emphasis is now being moved (and rightly so) to OOAD and OO development using integrated CASE (I-CASE) tools involving automatic code generation. Ideally, we require OO-ICASE tools with repositories for storing re-usable classes. These OO-ICASE tools need not be tied to OOPLs (though such linkages are encouraged), but should facilitate all or most aspects of OOSE, with automatic code generation.

Non-OO-CASE tools and 4GLs are many and varied. They remain useful aids in software development. Of course, fourth-generation languages (4GLs) based on non-procedural programming languages are preferred to those that use procedural languages.

A7.2 Categories of CASE Tools

As you are aware, a CASE tool is a sophisticated software product that assists in the development of other software products in a significant way. The CASE tool provides the following functionalities:

- Graphical user interface (GUI) so the user can model the system in a user-friendly environment
- Automatic code generation in at least one programming language (preferably an OOPL)
- Executable diagrams and integration of the GUI with the actual code generator
- Seamless integration/connectivity to various databases
- Facility to produce systems in deployment mode (consisting of executables of system resources without the bulk of the

CASE tool itself)

- A repository for defining and storing generic and/or reusable system components

CASE tools may be categorized from two broad perspectives:

- Supporting Technology
- Scope

In terms of supporting technology, there are three classifications:

- Non-OO-CASE Tool
- Pure-OO-CASE Tool
- Hybrid CASE Tool

In respect to scope, there are also three classifications:

- Fragmented CASE — supporting only a portion of the system life cycle
- Integrated CASE (I-CASE) — supporting all aspects of the SDLC
- Information Engineering CASE (IE-CASE) — supporting the whole process of information engineering for the entire enterprise

[Figure A7-1](#) illustrates the categorizations. In examining software development tools for subsequent development, products will fit (some vaguely) in various cells of the matrix. Prudent software evaluation will aid the selection process.

		Non-OO	Traditional with OO Flavor	Hybrid	Pure OO
Fragmented CASE	Front-end (upper)				
	Back-end (lower)				
I-CASE (but not IE)					
IE-CASE					

↑
Ideal Choice

Figure A7-1. Categories of CASE Tools

Recall that in [Figure 2-5](#) of chapter 2, several CASE tools were listed. [Figure A7-2](#) pulls some of these products and places them in the categories they best fit in (bear in mind that there is hardly any product which fits perfectly into a specific cell of the categories grid above).

Product	Classification
LANSA	Non-OO CASE Tool
LiveModel, Rational Rose, TogetherSoft	OO-ICASE Tool
Delphi, Team Developer, Java NetBeans	OO-CASE Tool; Front-end RAD Tool
DB2, Oracle, Sybase, Informix	Universal DBMS

Figure A7-2. Examples of Popular Products

A7.3 Universal Database Management Systems

In an effort to keep abreast with the advances in OOM, we have seen an increase in the upgrade of traditionally relational database management systems (RDBMS) into *object-oriented database management systems* (OO-DBMS) suites. Other terms used to describe these products are *universal database management system* (UDBMS) suites, or simply *object databases*. The rationale for these hybrid products are due to two important perspectives:

- Relational databases dominate the world of databases (and will continue to do so for the foreseeable future). Apart from the fact that the relational database model is very rigorous

and has been proven over the years, there are numerous circumstances where a relational database represents the most viable solution.

- The superiority of the OO model (particularly from a user interface perspective) is generally accepted by the industry.

UDBMS suites constitute a reasonable compromise:

- Companies that have dominated the market place with RDBMS can maintain their market share.
- Legacy systems can be integrated into the “world of objects” by simply “wearing” an object-oriented wrapper (user interface).
- Object-oriented technology, which clearly will dominate future software, will not make obsolete, the huge investments that have been made in relational systems. The best we can expect is a peaceful coexistence of both technologies, with each complementing the other.
- Many so-called OO software development tools are only object-oriented to the point of the user interface development. The typical scenario therefore, is to have an OO user interface superimposed on a relational database. Additionally, most of the front-end CASE tools and RAD (rapid application development) tools available facilitate this approach. Examples include (but are not confined to) Java NetBeans, Delphi, C++ Builder, Visual Studio, Team Developer, etc.

The universal database supports both the relational database as well as the OO database. The designer can therefore make critical decisions as to which approach is preferred, given the scenario. Several of the leading software engineering companies have in recent times, introduced such products. Examples include DB2, Oracle, and Informix.

A7.4 Benefits of OO-CASE Tools and

UDBMS Suites

Object technology relies heavily and (and typically presumes) the presence on OO-CASE tools to support the principles and techniques. The benefits of object technology (as discussed in [appendix 1](#)) are therefore applicable here.

Nonetheless, [Figure A7-3](#) underscores the significant benefits of UDBMS suites and [Figure A7-4](#) underscores the benefits of OO-CASE tools.

Leverage: The strengths of the relational model and the object model can be emphasized and the respective weak points avoided.

Legacy Systems: Legacy systems can be facilitated, since a relational database and an object oriented user interface will peacefully coexist.

Convenience: The benefits of a relational database and an OO application development are available to the organization.

Figure A7-3. Benefits of Universal Databases

Logical support for OOSE: As mentioned earlier, OO-CASE is a perfect facilitator for OOSE.

Simplified SDLC: The use of OO-CASE tools facilitates a significant change in system life cycle from several disjoint phases to two integrated, interwoven stages relating to the structure and behavior of objects (review appendices 1 and 4).

Faster System Development: By using O-CASE we experience a significant overall reduction in system development (by a factor of 40-80 percent). In fact, OO-CASE promotes RAD, so that most RAD tools are object oriented to some degree.

Powerful System Integration: OO-CASE tools facilitate a change from line-by-line system development to chunk-by-chunk (component-by-component) system development, with the facility to integrate previously built components from different systems. The CASE toolset typically includes a repository where resources (components, classes, and methods) to be used in different systems can be stored.

More Reliable Systems: With OO-CASE tools that generate code automatically, more reliable code is produced, leading to more reliable systems.

More Maintainable Systems: Systems developed using OO-CASE tools are much easier to maintain than systems developed by other means.

Figure A7-4. Significant Benefits of OO-CASE Tools

A7.5 Object Oriented Programming

Languages

An *object-oriented programming language* (OOPL) is a high level language that supports object-oriented programming. The language must support the fundamental principles of OO programming such as classes, inheritance, polymorphism, encapsulation, etc. (as discussed in [appendix 2](#)) in order to qualify as an OOPL. The important features to consider when choosing an OOPL are summarized in [Figure A7-5](#).

1. The OOPL should support fundamental principles of object technology as discussed in appendix 2 (classes, inheritance, polymorphism, encapsulation, etc).
2. Is the OOPL interpretive or a compiling language?
 - An interpretive language employs an *interpreter* to convert source code to object code. Translation is done on a line-by-line or command-by-command basis. This is very convenient for the developer, but is not as efficient as compiling languages. The reason for this is that the source code is translated each time the program is executed.
 - A compiling language employs a *compiler* to convert source code to object code. The object code is saved as an executable file for subsequent execution of the program. A compiling language does not provide the convenience that an interpretive language provides because the compiling process is a batch process (which produces a result after the entire program is analyzed).
 - *Dynamic compilation* is a compromise between the previous two approaches. Only modified methods (of an application) are recompiled. Also, if the user (programmer) tries to run a program, but there are methods that have been modified since the last compilation, a new compilation is automatically effected (without user intervention) and then (assuming no errors) the program is executed.
3. The OOPL should support pointers or dynamic arrays. Object technology is predicated on the use of pointers; without them, the whole idea breaks down.
4. The OOPL should support *late (dynamic) binding* as opposed to static binding. The process of determining which object another object references (points to) is called binding. Binding identifies the receiver of a request; it can be done at compilation time, but preferably at execution time. Compilation time binding is also called early binding or static binding. Execution time binding is also called late binding or dynamic binding.

[Figure A7-5. Desirable Features of OOPLs](#)

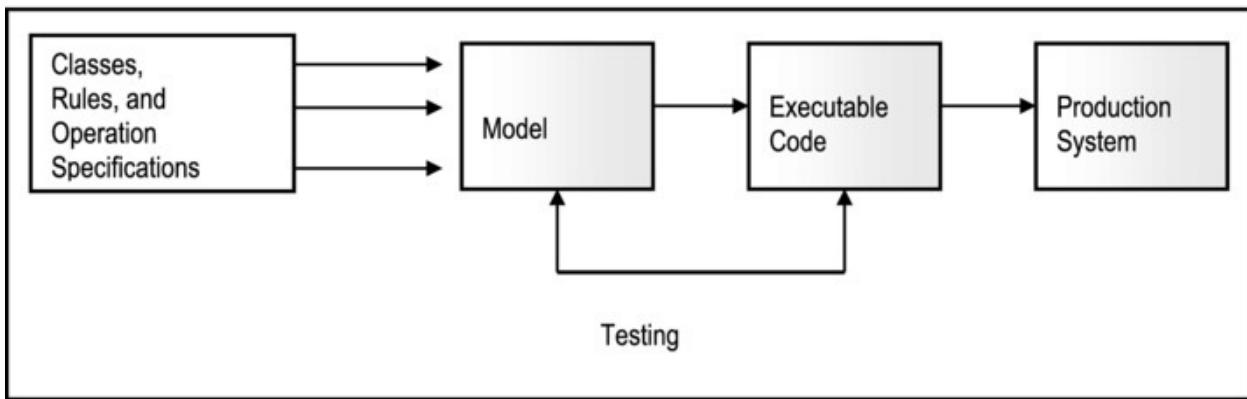
There are two categories of OOPLs: pure OOPLs and hybrid languages. Pure OOPLs are languages which from the outset, were designed to be object oriented; they do not support procedural programming at all. Examples of pure OOPLs include SmallTalk, Actor, Eiffel, Java, and Scalable Application Language (SAL).

Hybrid OOPLs are languages that have resulted from the upgrade of traditionally procedural languages. They support both procedural and object oriented programming. Examples include C++ (from C), Objective C (from C),

Visual Basic (from Basic), Object Pascal (from Pascal), Object COBOL (from COBOL), and CLOS (from LISP).

A7.6 Modeling and Code Generation

With the use of OO-CASE tools, the software development life cycle (SDLC) is reduced from several disjoint phases to two integrated, interwoven stages — modeling and code generation — as illustrated in [Figure A7-6](#).



[Figure A7-6. OO Modeling and Code Generation](#)

OO-CASE tools need not use OOPLs. They can generate code in portable HLLs, e.g. C, COBOL, Pascal, etc. One major problem with this, however is that most traditional HLL compilers do not support dynamic (late) binding. In order for the OO-CASE tool to be successful, the underlying programming language must support late binding. For this reason, OO-CASE tools usually generate code in hybrid languages (such as C++ and Object Pascal) or pure OO languages (such as Java and SAL).

In the future, the industry should be able to produce more OO-CASE tools that generate low or intermediate level code that is machine and/or compiler independent. In this regard, the Java revolution is promising.

A guiding principle of I-CASE is that debugging must not be at the code level, but at the higher modeling level that drives the code generation. Some products meet this criterion better than others. Most products usually provide the designer with the flexibility of deciding whether he/she wants to modify generated code or change the design.

A7.6.1 Instant CASE

Very sophisticated OO-CASE tools not only generate code as soon as classes are defined, and rules specified, but also create instantiation, so the software engineer can see and test the effect of his/her work with actual data. These are referred to as instant CASE tools. The effect is similar to that of a spreadsheet, except that it is much more sophisticated.

Instant CASE is still an ambition (and guiding principle) for many OO-CASE tool developers. However, a number of products have begun to provide this feature. Examples include LiveModel, Oracle, and Delphi. It is anticipated that this trend will continue.

A7.6.2 Repository

The OO-CASE tools should provide for the maintenance of a repository, as an integral part of its environment. Typically, the repository stores and catalogs reusable resources such as generic subsystems, classes, interfaces, methods, and data models. It also stores important information about different systems of the enterprise. In short, any item that is reusable across multiple subsystems or systems is a candidate for the repository.

The repository stores object types such as (but not confined to):

- Components
- Interfaces
- Classes
- Event types
- Rules
- Conditions
- Operations
- Data models
- Screen designs
- Report designs

A well-designed OO-CASE tool with repository is usually a useful facility

towards information engineering. For this and other reasons, CASE tools with repositories have become the norm in the software engineering industry.

Software systems should be designed with the use of a repository in mind. There is no replacement for good design! Moreover, as emphasized throughout this course, good design is not coincidental; rather, it is a consequence of deliberate effort.

A7.7 Standards for OOM

With the proliferation of tools that support OOM, it has become necessary to standardize these tools so that their usefulness to the consuming public can be maximized. Three prominent standards have emerged – CORBA, COM and .NET. A brief summary of each follows.

A7.7.1 CORBA

Since the start of the 1990s, the Object Management Group (OMG) has been attempting to establish standards for OO software. Since 1996, the OMG's Common Object Request Broker Architecture (CORBA) has advanced as the de facto industry standard.

The CORBA standards govern how objects communicate in a distributed system. They therefore affect distributed database systems and communication systems. CORBA is well documented and is available via the Worldwide Web (see recommended readings).

A7.7.2 COM

Another important standard is the Component Object Model (COM). Developed by Microsoft, there are two main enhancements of COM: Distributed COM (DCOM) and COM+. Then there are other COM related standards such as Microsoft Transaction Server (MTS) and ActiveX. These standards also support inter-object communication in distributed environments.

It is recommended that contemporary OO-CASE tools and DBMS suites support the CORBA and/or COM standards if they are to remain marketable.

A7.7.3 The .NET Boom

A third wave of technology craze is the .NET craze. This is a set of standards that facilitate easy communication among Microsoft products. These standards were recently developed by Microsoft, and have been off to a good start towards industry acceptance. The .NET products also facilitate easy Web accessibility. Further information can be obtained from the Microsoft .NET Web site (see recommended readings).

A7.8 Summary and Concluding Remarks

Let us summarize what we have covered in this chapter:

- Object technology and CASE technology fit naturally together.
- There are different categories of CASE tools — from object-oriented to procedural; they may be front-end, back-end, or integrated; they may or may not support information engineering.
- A UDBMS supports relational as well as object databases.
- OO-CASE tools and UDBMS suites provide a number of significant benefits to the software engineering industry.
- OOPLs come in two categories, namely pure OOPL and hybrid OOPL.
- Ideally, the OO-CASE tool used should support an OOPL. Additionally, the OO-CASE tool used should preferably be an instant CASE tool with repository support.
- Contemporary OO-CASE tools and DBMS suites must support the CORBA and/or COM standards if they are to remain marketable. The .NET standards are also emerging with increasing industry acceptance.

Care should be taken in choosing the OO tools used for your information engineering or software engineering project, as this could have significant effects on the outcome. By using the appropriate OO-CASE tools, you can reduce your project duration time by 20% – 60% of the time it would have taken without such tools.

A7.9 References and/or Recommended Readings

[Martin, 1993] Martin, James and James Odell. *Principles of Object Oriented Analysis and Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1993. See [chapters 17-19](#).

[Rumbaugh, 1991] Rumbaugh, James, et. al. *Object Oriented Modeling And Design*. Eaglewood Cliffs, NJ: Prentice Hall, 1991. See [chapters 15-17](#).

[CORBA, 1997-2009] CORBA. <http://www.corba.org> (accessed June 2008).

[OMG, 1997-2008] Object Management Group. <http://www.omg.org> (accessed June 2008).

[Microsoft, 2008a] Microsoft. *Microsoft.NET*.
<http://www.microsoft.com/.NET> (accessed June 2008).

[Microsoft, 2008b] Microsoft. *DCOM Architecture*.
<http://msdn.microsoft.com/en-us/library/ms809311.aspx> (accessed June 2008).

[Washington University, 2000-2007] Computer Science Department, Washington University. *Overview of CORBA*.
<http://www.cs.wustl.edu/~schmidt/corba-overview.html> (accessed June 2008).

APPENDIX 8



Project Proposal for a Generic Inventory Management System

This appendix provides excerpts from a sample project proposal (also called initial system requirement) for a generic Inventory Management System that may be suitable for a small or medium sized organization.

The document includes the following:

- Problem Definition
- Proposed Solution
- Scope of the System
- System Objectives
- Expected Benefits
- Overview of Storage Requirements
- Anticipated Outputs
- Feasibility Analysis Report
- Initial Project Schedule

A8.1 Problem Definition

When operating a business, an inventory management system is one of the fundamental needs. Inventory management is critical to any business. The absence of such a system could result in several problems, some of which are stated below:

1. If a business does not know what items are in stock it might lose significant sale opportunities. On the other hand if the inventory system shows items in stock that are actually not in stock then that is an embarrassing situation where a customer has to wait while the missing (non-existent) item is being located. This lack of professionalism could most certainly drive customers away.
2. In addition to this, there is also the problem of knowing exactly how much inventory should be kept on hand. Keeping too much inventory on hand is costly, resulting in storage costs, insurance costs, and salaries of individuals to manage/locate inventory. On the other hand, not having sufficient stock on hand is also costly, in that the business could lose significant opportunities for additional sale.
3. Knowing exactly how much stock is on hand is crucial to knowing exactly when items need to be reordered.
4. Inability to properly manage the inventory could severely affect the productivity and profitability of the company.

A8.2 Proposed Solution

This project seeks to address this problem by developing generic Inventory Management System (IMS) that is tailored to the specific needs of any company.

This software system must be portable and platform independent (if possible). In light of this, the recommended software development tools include Delphi or Java NetBeans on the front-end, and MySQL as the back-end DBMS.

A8.3 Scope of the System

The main components of the system will be as follows:

- A relational database in MySQL or another portable DBMS

- stores the essential data for subsequent analysis
- An Acquisitions Subsystem (AS) facilitates tracking of purchases, utilization and sale of inventory items
- A Financial Management Subsystem (FMS) provides the financial implications of various transactions
- Point of Sale Subsystem (POSS) manages interfacing between the point of sale equipment (barcode scanners, and invoice printers, etc.) and the internal database

A8.4 System Objectives

The Inventory Management System (IMS) will fulfill the following objectives:

- Accurate tracking of inventory
- Features to facilitate entry and update of inventory items and modify products and quantities
- Tracking of customers and suppliers contact information
- Tracking of outgoing purchase orders to suppliers
- Tracking of incoming purchase orders from customers
- Features to facilitate management of purchases and sale of goods
- Features to facilitate and handle return of goods purchased and sold
- Features to query various aspects of the company's inventory (e.g. lookup quantities for specific items) and the financial implications of this
- Facility to generate an inventory sheet
- Alerts for the reorder of inventory items
- Basic financial management capabilities which will give an overview of the financial status of the company (basic account receivable and accounts payable)

A8.5 Expected Benefits

The IMS will bring a number of benefits to the organization:

- The system will help to keep accurate records of inventory. This will be an invaluable aid to sales as well as purchasing.
- The maintenance of accurate records should reduce the likelihood of inventory loss or misappropriation.
- The system will reduce the amount of human effort needed, and hence the overhead cost of the business.
- The system will contribute to improving the public image of the business — customers will appreciate being able to know instantly exactly what is available without having to waste time.
- The system should facilitate better management decisions, as well as improved productivity.
- The system will facilitate better management of the company's resources. For instance, instead of having money tied up in unnecessary inventory, the company can have a just-in-time (JIT) delivery system, and thereby free up funds that would otherwise be used in inventory storage, to be available for other aspects of the business.

A8.6 Overview of Storage Requirements

It is anticipated that the system will contain the following main information entities:

- Inventory Master
- Item Categories
- Suppliers

- States or Provinces
- Customers
- Purchase Orders
- Purchase Invoices
- Purchase Returns
- Sale Orders
- Sale Invoices
- Sale Returns
- Chart of Accounts
- Accounting Balances
- Purchase Invoice Payment Plans
- Payments Made
- Financial Institutions of Interest
- Sale Invoice Payment Plans
- Payments Received
- Financial Transactions
- Investments

A8.7 Anticipated Outputs

It will be possible to run queries and obtain reports on all information stored. As such, operations will be provided to extract and display information from the above mentioned information entities. Some of the more prominent outputs are as follows:

- Inventory Master Listings
- Sales Orders
- Sales Invoices
- Payments Received Log

- Accounts Receivable
- Returns History
- Purchase Orders
- Purchase Invoices
- Payments Made Log
- Accounts Payable
- Account Balances — Accounts Payable and Accounts Receivable
- Financial Transactions Log
- Investments Log

A8.8 Feasibility Analysis Report

Possible alternate solutions examined are

- Alternative A: Manual system where inventory is kept track of by hand
- Alternative B: Buy an off-the-shelf solution and customize it
- Alternative C: Contract a software engineering firm to develop the system
- Alternative D: Develop system as a student project (for the purpose of illustration and academic research, this alternative will be taken)

A8.8.1 Feasibility of Alternative A

The **technical feasibility** of this alternative may be summarized as follows:

- No expensive computer equipment would be required; only folders, files, and basic office equipment would be needed.
- No computer software would be needed.

- Not much knowledge and expertise is required since this kind of job would be basically filing forms and keeping track of inventory.
- No other technology would be required.

The **economic feasibility** of this alternative may be summarized as follows:

- The cost of this option is minimal (the wages of the employees who count stock and fill out the inventory forms).
- The cost of the office supplies needed for this system would be minimal.

The **operational feasibility** of this alternative may be summarized as follows:

- This method of having to manually count stock would be time consuming and painstakingly difficult and it offers no guarantee of success.
- There would be no development time since this is a manual system.
- Implementation time would also be low — because this manual system is easy to learn not much training is required to be able to do the job.

The **risks** associated with this approach are extremely high. All the problems stated above would remain. This could potentially run the business into bankruptcy.

A8.8.2 Feasibility of Alternative B

The **technical feasibility** of this alternative may be summarized as follows:

- The business no doubt already has the computer hardware that would be required to run the software.
- Customizing the software package could be tedious if the acquired software system is poorly designed.

- Maintaining the software could be problematic, especially if the system is poorly designed, and the documentation is scanty and/or inadequate.
- There is no guarantee that the purchased software system will be comprehensive enough.

In terms of **economic feasibility**, the initial investment may be affordable. However, estimating maintenance cost could be tricky.

The **operational feasibility** of this alternative may be summarized as follows:

- There would be no development time since the software is already made.
- The implementation time would be related to how well-designed the software system is. If it's well designed, installation time and time to learn the software should be minimal.
- User training would be required.

The **risks** associated with this approach are high, since there is no guarantee that software acquired will live up to expectations.

A8.8.3 Feasibility of Alternative C

The **technical feasibility** of this alternative may be summarized as follows:

- Acquiring the required software development environment would be the responsibility of the contracted company.
- Knowledge and expertise might be required in learning the new software and how to use and apply it, but a well-developed software system that is custom-made for the business should be easy to learn.
- The developers themselves would have to spend some time learning about the operation and needs of the business in order to be able to develop a suitable software system.

The **operational feasibility** of this alternative may be summarized as follows:

- The initial investment would be very high because paying a software engineering company for development time can be very costly.
- Development time could be as much as 6 months to a year, but for an experienced software engineering firm, it shouldn't take more than a few months.
- The economic life of such custom-made software should be relatively long if the software is well made and well designed and if the needs of the business don't change drastically, in which case the system would either be not needed or wouldn't be adequate enough for the new needs.

The **operational feasibility** of this alternative may be summarized as follows:

- User training would be required.
- Implementation time should be short, provided that the system is adequately designed.

The **risks** associated with this approach could be high or low, depending on how the project is managed. There is no guarantee that software engineering firm will deliver a good job; however, precautions can be taken to ensure software quality.

A8.8.4 Feasibility of Alternative D

The **technical feasibility** of this alternative may be summarized as follows:

- Acquisition of the required software development environment would be required.
- Required knowledge and expertise are readily available.

The **operational feasibility** of this alternative may be summarized as follows:

- Technology required is relatively inexpensive.
- The development time would be high, but the benefits gained from the experience would be huge.

The **operational feasibility** of this alternative may be summarized as follows:

- The implementation time would be long, mainly because the software would have to be developed by student effort.
- Minimal user training would be required.

The **risks** associated with this approach could be high or low, depending on how the project is administered. The student has total control over who works on the project, so depending on the quality of the project team risks may or may not be reduced.

A8.8.5 Evaluation of Alternatives

The following table provides a comparison of the project alternatives, based on a number of feasibility factors. For each factor, a rank in the range {0 .. 20} is given for each alternative.

Feasibility Factors		Alt-A	Alt-B	Alt-C	Alt-D
Technical Feasibility		68	60	78	78
Availability of Hardware (bigger means better)	[Max 20]	20	15	20	20
Availability of Software (bigger means better)	[Max 20]	18	15	20	20
Availability of Expertise (bigger means better)	[Max 20]	14	16	20	20
Availability of Technology (bigger means better)	[Max 20]	16	12	18	18
Economic Feasibility		106	107	108	108
Engineering Cost (bigger means lower cost)	[Max 20]	18	17	17	17
Equipment Cost (bigger means lower cost)	[Max 20]	17	17	17	17
Operational Cost (bigger means lower cost)	[Max 20]	19	19	19	19
Facilities Cost (bigger means lower cost)	[Max 20]	19	19	19	19
Development Time (bigger means shorter time)	[Max 20]	18	18	19	19
Economic Life (bigger means longer time)	[Max 20]	16	18	18	18
Risk (bigger means lower risk)	[Max 20]	18	18	18	18
Operational Feasibility		50	52	56	56
User Attitude to Likely Changes (bigger means better)	[Max 20]	20	16	18	18
Likelihood of Organizational Policy Changes (bigger means fewer changes)	[Max 20]	16	19	18	18
Implementation Time (bigger means shorter time)	[Max 20]	14	17	20	20
Software Quality		16	110	176	176
Maintainability (bigger means better)	[Max 20]	00	10	16	16
Efficiency (bigger means better)	[Max 20]	04	10	16	16
User Friendliness (bigger means better)	[Max 20]	00	10	16	16
Documentation (bigger means better)	[Max 20]	04	10	16	16
Compatibility (bigger means better)	[Max 20]	00	10	16	16
Security (bigger means better)	[Max 20]	04	10	16	16
Reliability (bigger means better)	[Max 20]	02	10	16	16
Flexibility & Functionality (bigger means better)	[Max 20]	00	10	16	16
Adaptability (bigger means better)	[Max 20]	00	10	16	16
Growth Potential (bigger means better)	[Max 20]	02	10	16	16
Productivity (bigger means better)	[Max 20]	02	10	16	16
Overall Evaluation Score	[Max 500]	240	329	418	418
Note:					
1.	Feasibility factors were evaluated based on the foregoing discussion for each alternative.				
2.	Quality factors were evaluated based on estimates of the performance of each alternative.				
3.	Based on this analysis, alternative D has been selected as the most prudent.				

Figure A8-1. Feasibility Evaluation Grid Showing Comparison of System Alternatives

A8.9 Initial Project Schedule

Figure A8-2 shows an initial project schedule. This will be further refined as more details become available. The estimated duration (assuming three full-time software engineers) is 41 weeks.

Activity #	Activity Description	Weeks
1	Initial System Requirement	2
2	Requirements Specification	6
3	Design Specification	10
4	Database Creation	2
5	Software Development	14
6	Software Testing	2
7	Software Documentation	4
8	Software Installation & Delivery	1
Total Estimated Duration		41
Assumption: Project team of three software engineers		

Figure A8-2. Initial Project Schedule

APPENDIX 9



Requirements Specification for a Generic Inventory Management System

This appendix provides excerpts from a sample requirements specification (RS) for a generic Inventory Management System (IMS) that may be suitable for a small or medium sized organization. The document is a follow-up to the ISR of [Appendix 8](#), and includes the following:

- System Overview
- Storage Requirements
- Operational Requirements
- Business Rules
- Summary and Concluding Remarks

A9.1 System Overview

A9.1.1 Problem Definition

See section A8.1 of [Appendix 8](#).

A9.1.2 Proposed Solution

See section A8.2 of [Appendix 8](#).

A9.1.3 System Architecture

The System will have four main components:

- Acquisitions Management Subsystem (Java or Delphi)
- Financial Management Subsystem (Java or Delphi)
- Point of Sale Subsystem (POSS)
- Database Backbone (MySQL)

[Figure A9-1](#) provides the information topology chart (ITC) and [Figure A9-2](#) shows the object flow diagram (OFD).

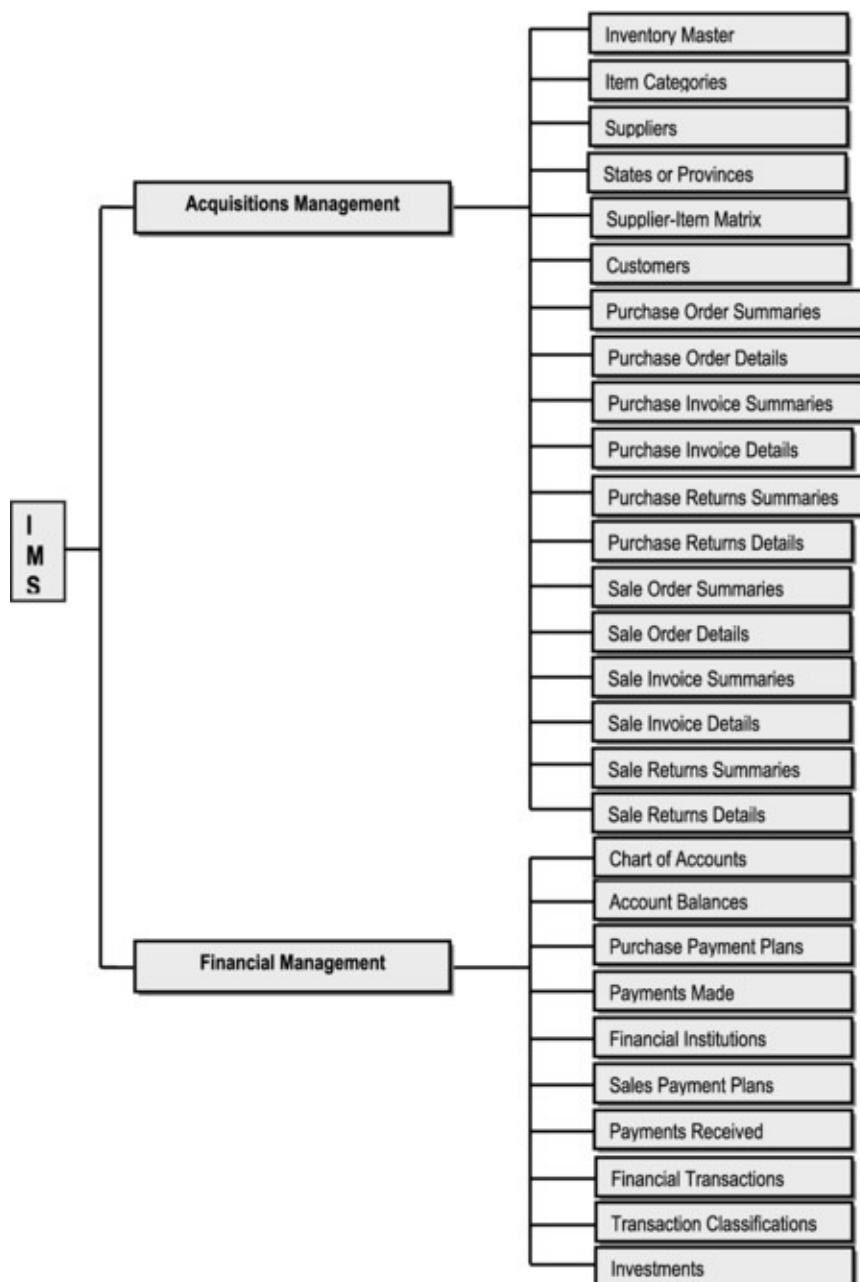


Figure A9-1. IMS Information Topology Chart

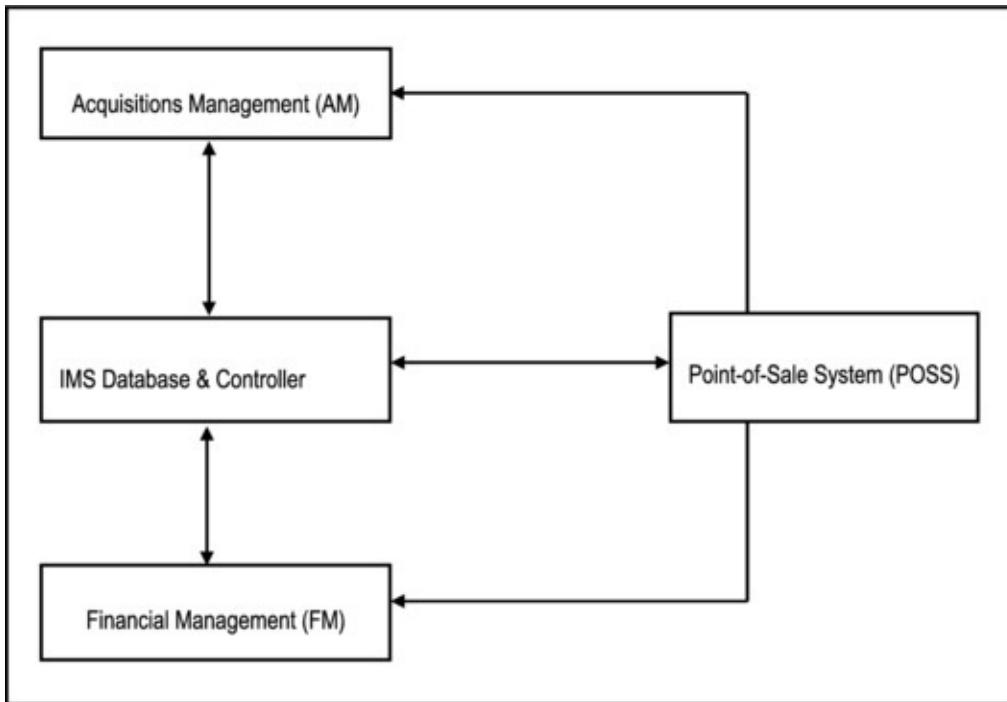


Figure A9-2. IMS Object Flow Diagram

A9.2 Storage Requirements

The main storage requirements of the system will be covered in this chapter. The conventions used are summarized below:

- Each information entity referenced is identified by a reference code and a descriptive name.
- For each entity the attributes (data elements) to be stored are identified.
- The entities as presented, will easily transition into a set of normalized relations in a normalized relational database.
- Data elements that will be implemented as foreign keys, in the normalized relational database, are identified by an asterisk (*) followed by a parenthesized comment, specifying what entity they reference.
- Data elements that will be used as primary key attributes (or part of the primary key) at database implementation time are

also identified by parenthesized comments.

- For each entity a comment describing the data to be stored is provided.

The Acquisitions Management Subsystem will be addressed first, followed by the Financial Management Subsystem.

A9.2.1 Acquisitions Management Subsystem

[Figure A9-3](#) provides storage specifications for the main information entities comprising the Acquisitions Management Subsystem.

E1. Inventory Master consists of the following attributes:	
Attributes:	
1. Item Code [Key] 2. Item Name / Description 3. Item Category * (Refers to E2) 4. Quantity on Hand 5. Reorder Quantity 6. Last Unit Price 7. Average Unit Price 8. Last Selling Price 9. Current Selling Price 10. Account Number * (refers to E18) 11. UPC Code 12. SKU Number	
Comments:	This entity stores information about Inventory items.

E2. Item Category consists of the following attributes:

Attributes:

1. Category Code [Key]
2. Category Description

Comments:

This entity stores information about Item Categories

E3. Supplier consists of the following attributes:

Attributes:

1. Supplier Code [Key]
2. Supplier Name
3. Address Line 1
4. Address Line 2
5. State or Province Code * (Refers to E4)
6. Zip Code
7. Telephone Number(s)
8. Fax Number
9. Email Address
10. Contact Person
11. Account Number * (Refers to E18)
12. Ordering Preference

Comments:

This entity stores information about Suppliers

E4. State or Province consists of the following attributes:

Attributes:

1. State / Province Code [Key]
2. State / Province Name

Comments:

This entity stores the List of States and Provinces

E5. Supplier-Item Matrix consists of the following attributes:

Attributes:

1. Supplier Code * (Refers to E3) [Key 1]
2. Item Code * (Refers to E1) [Key 2]

Comments:

This entity stores information about which items come from which Supplier

E6. Customer consists of the following attributes:

Attributes:

1. Customer Code [Key]
2. Customer Name
3. Address Line 1
4. Address Line 2
5. State / Province Code * (Refers to E4)
6. Zip Code
7. Telephone Number(s)
8. Fax Number (s)
9. Email Address
10. Contact Person
11. Account Number * (Refers to E18)
12. Billing Preferences

Comments:

This entity stores information about Customers

E7. Purchase Order Summary consists of the following attributes:

Attributes:

1. Purchase Order Number [Key]
2. Purchase Order Supplier Code * (Refers to E3)
3. Purchase Order Date
4. Purchase Order Status (Filled/ Partial/ Outstanding)
5. Purchase Order Estimated Amount
6. Purchase Order Estimated Discount

Comments:

This entity stores information about Purchase Orders

E8. Purchase Order Detail consists of the following attributes:

Attributes:

1. Purchase Order Number * (Refers to E7) [Key 1]
2. Item Code * (Refers to E1) [Key 2]
3. Quantity Ordered
4. Order Unit Price

Comments:

This entity stores information about Purchase Order Details

E9. Purchase Invoice consists of the following attributes:

Attributes:

1. Invoice Number [Alternate Key 1]
2. Invoice Supplier Code * (Refers to E3) [Alternate Key 2]
3. Invoice Date [Alternate Key 3]
4. Related Purchase Order Number * (Refers to E7)
5. Invoice Amount
6. Invoice Amount Outstanding
7. Discount
8. Tax
9. Comment
10. Purchase Invoice Identification Number [Primary Key]
11. Transaction Reference Number * (Refers to E26)

Comments:

This entity stores information about Purchase Invoices

E10. Purchase Invoice Detail consists of the following attributes:

Attributes:

1. Purchase Invoice Identification * (Refers to E9) [Key 1]
2. Item Number * (Refers to E1) [Key 2]
3. Item Quantity
4. Item Unit Price

Comments:

This entity stores information about Purchase Invoice Details

E11. Purchase Returns Summary consists of the following attributes:

Attributes:

1. Purchase Invoice Identification * (Refers to E9)
2. Return Date
3. Return Amount
4. Purchase Return ID [Key]
5. Transaction Reference Number * (Refers to E26)

Comments:

This entity stores information about Purchase Returns

E12. Purchase>Returns Detail consists of the following attributes:

Attributes:

1. Purchase Return ID * (Refers to E11) [Key 1]
2. Item Code * (Refers to E1) [Key 2]
3. Quantity Returned
4. Return Unit Price
5. Comment

Comments:

This entity stores information about Purchase Return Details

E13. Sale Order Summary consists of the following attributes:

Attributes:

1. Sales Order Number [Alternate Key 1]
2. Sales Order Customer Code * (Refers to E6) [Alternate Key 2]
3. Sales Order Date [Alternate Key 3]
4. Sales Order Status (Filled/ Partial/ Outstanding)
5. Sales Order Estimated Amount
6. Sales Order Estimated Discount
7. Sales Order Identification Number [Primary Key]

Comments:

This entity stores information about Sales Orders

E14. Sales Order Detail consists of the following attributes:

Attributes:

1. Sales Order Identification Number * (Refers to E13)
2. Item Code * (Refers to E1)
3. Quantity Ordered
4. Anticipated Unit Price

Comments:

This entity stores information about Sales Order Details

E15. Sales Invoice Summary consists of the following attributes:

Attributes:

1. Sales Invoice Number [Alternate Key 1]
2. Sales Invoice Customer Code * (Refers to E6) [Alternate Key 2]
3. Sales Invoice Date [Alternate Key 3]
4. Related Sale Order ID * (Refers to E13)
5. Sales Invoice Amount
6. Sales Invoice Amount Outstanding
7. Sales Invoice Discount
8. Sales Invoice Tax
9. Sales Invoice Comment
10. Sales Invoice Identification Number [Primary Key]
11. Transaction Reference Number * (Refers to E26)

Comments:

This entity stores information about Sales Invoices

E16. Sales Invoice Detail consists of the following attributes:

Attributes:

1. Sales Invoice Identification Number * (Refers to E15) [Key 1]
2. Item Code * (Refers to E1) [Key 2]
3. Item Quantity
4. Item Unit Price

Comments:

This entity stores information about Sales Invoice Details

E17. Sales Return Summary consists of the following attributes:
Attributes:
1. Sales Invoice Identification Number * (Refers to E15)
2. Return Date
3. Return Amount
4. Sale Return ID [Key]
5. Transaction Reference Number * (Refers to E26)
Comments:
This entity stores information about Sales Returns
E18. Sales Return Detail consists of the following attributes:
Attributes:
1. Sale Return ID * (Refers to E17) [Key 1]
2. Item Code * (Refers to E1) [Key 2]
3. Quantity Returned
4. Return Unit Price
5. Comment
Comments:
This entity stores information about Sales Returns

Figure A9-3. Storage Specifications for the Acquisitions Management Subsystem

A9.2.2 Financial Management Subsystem

[Figure A9-4](#) provides storage specifications for the Financial Management Subsystem.

E19. Chart of Accounts consists of the following attributes:**Attributes:**

1. Account Number [Key]
2. Account Description
3. Summary / Detail Flag
4. Parent Account * (Refers to E19)

Comments:

This entity stores information about the Various Accounts in the system

E20. Account Balance consists of the following attributes:**Attributes:**

1. Financial Period [Key 1]
2. Account Number * (Refers to E19) [Key 2]
3. Balance
4. Comment

Comments:

This entity stores information about the Account Balances

E21. Purchase Payment Plan consists of the following attributes:**Attributes:**

1. Purchase Invoice Identification Number * (Refers to E9) [Key]
2. Required Day of Month
3. Number of Payments Required
4. Payment Begin Date

Comments:

This entity stores information about Invoice Payment Plans

E22. Payments Made Log consists of the following attributes:**Attributes:**

1. Purchase Invoice Identification Number * (Refers to E9) [Alt. Key 1]
2. Payment Date [Alternate Key 2]
3. Amount Paid
4. Payment Type (Cash / Check / Credit Card)
5. Institution Code * (Refers to E23)
6. Check / Credit Account Number
7. Comment
8. Payment Identification Code [Primary Key]
9. Transaction Reference Number * (Refers to E26)

Comments:

This entity stores information about Payments Made

E23. Financial Institution consists of the following attributes:

Attributes:

1. Institution Code [Key]
2. Institution Name
3. Address Line 1
4. Address Line 2
5. State / Province Code * (Refers to E4)
6. Zip Code
7. Telephone Number(s)
8. Fax Number(s)
9. Contact Person
10. Email Address

Comments:

This entity stores information about Financial Institutions

E24. Sales Payment Plan consists of the following attributes:

Attributes:

1. Sales Invoice Identification Number * (Refers to E15) [Key]
2. Required Day of Month
3. Number of Payments Required
4. Payment Begin Date

Comments:

This entity stores information about Sales Invoices Payment Plans

E25. Payments Received Log consists of the following attributes:

Attributes:

1. Sales Invoice Identification Number * (Refers to E15) [Alt. Key 1]
2. Payment Received Date [Alternate Key 2]
3. Amount Received
4. Payment Type (Cash / Check / Credit Card)
5. Institution Code * (Refers to E23)
6. Check / Credit Account Number
7. Comment
8. Payment Identification Code [Primary Key]
9. Transaction Reference Number * (Refers to E26)

Comments:

This entity stores information about Payments Received

E26. Financial Transactions Log consists of the following attributes:

Attributes:

1. Transaction ID Number [Key]
2. Transaction Date
3. Transaction Classification Code * (Refers to E27)
4. Account Number * (Refers to E18)
5. Debit Amount
6. Credit Amount
7. Accounting Period
8. Transaction Reference Number (see Note below)

Comments:

This entity stores information about Financial Transactions

Note: Transaction Reference Number ties back to one of the following entities:

PI: Purchase Invoice Transactions (**E9**)

PR: Purchase Return Transactions (**E11**)

SI: Sales Invoice Transactions (**E15**)

SR: Sale Return Transactions (**E17**)

PP1: Cash Purchase Payments Transactions (**E22**)

PP2: Credit Purchase Payments Transactions (**E22**)

SP1: Cash Sales Payments Transactions (**E25**)

SP2: Credit Sales Payments Transactions (**E25**)

INV: Investment Transactions (**E28**)

Comments:

These will be implemented via logical views.

E27. Transaction Classification consists of the following attributes:

Attributes:

1. Transaction Classification Code [Key]
2. Transaction Classification Description

Note: Transaction Classifications include:

PI – Purchase Invoices

PR – Purchase Returns

SI – Sales Invoices

SR – Sales Returns

PP1 – Cash Purchase Payments

PP2 – Credit Purchase Payments

SP1 – Cash Sales Payments

SP2 – Credit Sales Payments

JED – Journal Entries Debit

JEC – Journal Entries Credit

INV – Investments

Comments:

This entity stores information about Transaction Classifications

E28. Investments Log consists of the following attributes:
Attributes:
<ol style="list-style-type: none"> 1. Investment ID Code [Key] 2. Investment Transaction Date 3. Investment Amount 4. Institution Code * (Refers to E23) 5. Institutional Account Number 6. Internal Account Number * (Refers to E18) 7. Comment 8. Transaction Reference Number * (Refers to E26)
Comments: This entity stores information about Investments

Figure A9-4. Storage Specifications for the Financial Management Subsystem

A9.3 Operational Requirements

The information entities of the previous chapter will be implemented as a relational database using MySQL or some other appropriate DBMS. Superimposed on this database will be an OO GUI consisting of various operations that provide functionality and flexibility to the end users.

The user functionality and flexibility will be provided based on the following approach:

- For each information entity, define an object type of the same name.
- For each object type, define a set of basic operations that will facilitate addition, modification, deletion, inquiry, and reporting of data related to that object type (note that each object type does not necessarily require all five basic operations).
- For selected object types, introduce flexibility and sophistication such as additional and/or more complex inquiry and/or report operations that facilitate end users specifying selection criteria of their choice.
- Define additional utility operations that will support the user

interface of the system.

Figure A9-5 shows an initial list of operations that will be defined on each object type. An operation specification will be provided for each of these operations in the project's design specification. The reference code for each operation is indicated in square brackets.

Acquisitions Management Subsystem	
Object Type	Operations
E1 Inventory Master	Add Inventory Item [E1_A] Modify Inventory Item [E1_M] Delete Inventory Item [E1_Z] Inquire Inventory Item [E1_I] Report Inventory Item [E1_R]
E2 Item Category	Add Item Category [E2_A] Modify Item Category [E2_M] Delete Item Category [E2_Z] Inquire Item Category [E2_I] Report Item Category [E2_R]
E3 Supplier	Add Supplier [E3_A] Modify Supplier [E3_M] Delete Supplier [E3_Z] Inquire Supplier [E3_I] Report [E3_R]
E4 State or Province	Add State or Province [E4_A] Delete State or Province [E4_Z] Inquire State or Province [E4_I]
E5 Supplier-Item Matrix	Add Supplier-Item Matrix [E5_A] Modify Supplier-Item Matrix [E5_M] Delete Supplier-Item Matrix [E5_Z] Inquire Supplier-Item Matrix [E5_I] Report Supplier-Item Matrix [E5_R]
E6 Customer	Add Customer [E6_A] Modify Customer [E6_M] Delete Customer [E6_Z] Inquire Customer [E6_I]
E7 Purchase Order Summary	Add Purchase Order [E7_A] Modify Purchase Order [E7_M] Delete Purchase Order [E7_Z] Inquire Purchase Order [E7_I] Report Purchase Order [E7_R]
E8 Purchase Order Detail	Add Purchase Order Detail [E8_A] Modify Purchase Order Detail [E8_M] Delete Purchase Order Detail [E8_Z] Inquire Purchase Order Detail [E8_I] Report Purchase Order Detail [E8_R]

Acquisitions Management Subsystem	
Object Type	Operations
E9 Purchase Invoice	Add Purchase Invoice [E9_A] Modify Purchase Invoice [E9_M] Delete Purchase Invoice [E9_Z] Inquire Purchase Invoice [E9_I] Report Purchase Invoice [E9_R]
E10 Purchase Invoice Detail	Add Purchase Invoice Detail [E10_A] Modify Purchase Invoice Detail [E10_M] Delete Purchase Invoice Detail [E10_Z] Inquire Purchase Invoice Detail [E10_I] Report Purchase Invoice Detail [E10_R]
E11 Purchase Returns Summary	Add Purchase Returns [E11_A] Delete Purchase Returns [E11_Z] Inquire Purchase Returns [E11_I] Report Purchase Returns [E11_R]
E12 Purchase Returns Detail	Add Purchase Returns Detail [E12_A] Modify Purchase Returns Detail [E12_M] Delete Purchase Returns Detail [E12_Z] Inquire Purchase Returns Detail [E12_I] Report Purchase Returns Detail [E12_R]
E13 Sale Order Summary	Add Sale Order [E13_A] Modify Sale Order [E13_M] Delete Sale Order [E13_Z] Inquire Sale Order [E13_I] Report Sale Order [E13_R]
E14 Sale Order Detail	Add Sale Order Detail [E14_A] Modify Sale Order Detail [E14_M] Delete Sale Order Detail [E14_Z] Inquire Sale Order Detail [E14_I] Report Sale Order Detail [E14_R]
E15 Sale Invoice Summary	Add Sale Invoice [E15_A] Modify Sale Invoice [E15_M] Delete Sale Invoice [E15_Z] Inquire Sale Invoice [E15_I] Report Sale Invoice [E15_R]
E16 Sale Invoice Detail	Add Sale Invoice Detail [E16_A] Modify Sale Invoice Detail [E16_M] Delete Sale Invoice Detail [E16_Z] Inquire Sale Invoice Detail [E16_I] Report Sale Invoice Detail [E16_R]
E17 Sale Returns Summary	Add Sale Returns [E17_A] Modify Sale Returns [E17_M] Delete Sale Returns [E17_Z] Inquire Sale Returns [E17_I] Report Sale Returns [E17_R]
E18 Sale Return Detail	Add Sale Return Detail [E18_A] Delete Sale Return Detail [E18_Z] Inquire Sale Return Detail [E18_I] Report Sale Return Detail [E18_R]

Financial Management Subsystem	
Object Type	Operations
E19 Chart of Accounts	Add Chart of Accounts [E19_A] Delete Chart of Accounts [E19_Z] Inquire Chart of Accounts [E19_I] Report Chart of Accounts [E19_R]
E20 Account Balances	Add Account Balances [E20_A] Modify Account Balances [E20_M] Delete Account Balances [E20_Z] Inquire Account Balances [E20_I] Report Account Balances [E20_R]
E21 Purchase Invoice Payment Plan	Add Purchase Invoice Payment Plan [E21_A] Modify Purchase Invoice Payment Plan [E21_M] Delete Purchase Invoice Payment Plan [E21_Z] Inquire Purchase Invoice Payment Plan [E21_I] Report Purchase Invoice Payment Plan [E21_R]
E22 Payments Made	Add Payments Made [E22_A] Delete Payments Made [E22_Z] Inquire Payments Made [E22_I] Report Payments Made [E22_R]
E23 Financial Institutions	Add Financial Institutions [E23_A] Delete Financial Institutions [E23_Z] Inquire Financial Institutions [E23_I] Report Financial Institutions [E23_R]
E24 Sales Invoice Payment Plan	Add Sales Invoice Payment Plan [E24_A] Delete Sales Invoice Payment Plan [E24_Z] Inquire Sales Invoice Payment Plan [E24_I] Report Sales Invoice Payment Plan [E24_R]
E25 Payments Received	Add Payments Received [E25_A] Delete Payments Received [E25_Z] Inquire Payments Received [E25_I] Report Payments Received [E25_R]
E26 Financial Transactions	Add Financial Transactions [E26_A] Modify Financial Transactions [E26_M] Delete Financial Transactions [E26_Z] Inquire Financial Transactions [E26_I] Report Financial Transactions [E26_R]
E27 Transaction Classifications	Add Transaction Classifications [E27_A] Delete Transaction Classifications [E27_Z] Inquire Transaction Classifications [E27_I] Report Transaction Classifications [E27_R]
E28 Investment	Add Investment [E28_A] Modify Investment [E28_M] Delete Investment [E28_Z] Inquire Investment [E28_I] Report Investment [E28_R]

Figure A9-5. IMS User Operations List

A9.4 Business Rules

A9.4.1 Overview

There are four types of business rules to be covered:

- Relationship Integrity Rules: These will not be emphasized here since most of them can be deduced from the Storage Requirements.
- Data Integrity Rules: Most of these can also be deduced from the Storage Requirements.
- Derivation Rules: These will be mentioned in this chapter.
- Procedural Rules: These will also be mentioned in this chapter.

A9.4.2 Derivation and Procedural Rules

The following derivation and procedural rules will be enforced:

1. **Supplier** and **Customer** entities will each store a record called **Miscellaneous** for **Cash Purchases** and **Cash Sales** respectively.
2. **Cash Purchases** will include the following updates:
 - a. Write a record to **Purchase Invoice Summary** (related Purchase Order is null). If the **Supplier** is not listed in the **Supplier** entity use the default **Miscellaneous Supplier** record and put a comment in the comment field.
 - b. Write corresponding detail records in **Purchase Invoice Detail**.

- c. Write a record in the **Payments Made** entity, for the relevant items.
- d. Adjust the **Quantity-on-Hand** in the **Inventory Master** entity.
- e. Write corresponding record in the **Financial Transactions Log**.

3. **Credit Purchases** will include the following updates:

- a. Write a record to **Purchase Invoice Summary**.
- b. Write corresponding records to **Purchase Invoice Detail**.
- c. Update the related **Purchase Order Summary** record.
- d. Adjust **Quantity-on-Hand** in **Inventory Master** entity for the relevant items.
- e. Write **Accounts Payable** entry in **Financial Transactions Log**.

4. **Payments Made** will include the following updates:

- a. Write a record to **Payments Made** entity.
- b. Update the **Invoice Outstanding Amount** in the **Purchase Invoice Summary** entity.
- c. Write a record in the **Financial Transactions Log**.

5. Each **Purchase Return** will include the following updates:

- a. Write a record to the **Purchase Returns Summary** entity.
- b. Write corresponding detail records to **Purchase Returns Detail**.
- c. Adjust the **Quantity-on-Hand** in the **Inventory Master** entity, for the relevant items.
- d. Adjust **Outstanding-Amount** in **Purchase Invoice Summary**.

- e. Write corresponding record in **Financial Transactions Log**.

6. **Cash Sales** will include the following updates:

- a. Write a record to the **Sales Invoice Summary**. If the customer is not listed in the **Customer** entity, use the default **Miscellaneous Customer**, with an appropriate comment in the comment field.
- b. Write corresponding detail records in the **Sales Invoice Detail**.
- c. Write a record in the **Payments Received** entity.
- d. Adjust the **Quantity-on-Hand** in the **Inventory Master** entity, for the relevant items.
- e. Issue a receipt and write record(s) in the **Financial Transactions Log**.

7. **Credit Sales** will include the following updates:

- a. Write a record to the **Sales Invoice Summary** entity.
- b. Write corresponding detail records in **Sales Invoice Detail**.
- c. Adjust the **Quantity-on-Hand** in the **Inventory Master** entity, for the relevant items.
- d. Write **Accounts Receivable** record in **Financial Transactions Log**.

8. **Payments Received** will include the following updates:

- a. Write a record to **Payments Received** entity.
- b. Update the **Invoice Outstanding Amount** in the **Sales Invoice Summary** entity.
- c. Issue a receipt.
- d. Write corresponding record in **Financial Transactions Log**.

9. **Sales Returns** will include the following changes:

- a. Write a record to the **Sales Returns Summary** log.
 - b. Write corresponding details records to **Sales Returns Detail** log.
 - c. Adjust the **Quantity-on-Hand** in the **Inventory Master** entity, for the relevant items.
 - d. Adjust the **Outstanding-Amount** for the related **Sale Invoice**.
 - e. Write corresponding record in **Financial Transactions Log**.
10. Accounts Payable is determined by the formula:

Accounts Payable = [Purchase Invoices with non-zero Outstanding Amounts minus Purchase Returns with non-zero Return Amounts]

11. Accounts Receivable is determined by the formula:

Accounts Receivable = [Sales Invoices with non-zero Outstanding Amounts minus Sales Returns with non-zero Return Amounts]

These rules will be extremely useful during system development. Some of them will be incorporated into the relevant operation specifications, and included in the project's design specification.

A9.5 Summary and Concluding Remarks

This document provided a basic blueprint for a generic Inventory Management System (IMS). It includes the following:

- An overview of the system, including problem definition, proposed solution, and system architecture.

- Storage requirements, including draft outlines for the basic information entities comprising the system.
- Operational requirements, including an initial list of user operations.
- Business rules, including derivation and procedural rules.

The design specification will use these requirements to prepare a more detailed set of specifications from which the system will be developed.

APPENDIX 10



Design Specification for a Generic Inventory Management System

This appendix provides excerpts from a sample design specification (DS) for a generic Inventory Management System (IMS) that may be suitable for a small or medium sized organization. The document is a follow-up to the RS of [appendix 9](#), and includes the following:

- System Overview
- Database Specification
- Operations Specification
- User Interface Specification
- Message and Help Specifications
- Summary and Concluding Remarks

A10.1 System Overview

A10.1.1 Problem Definition

See section A8.1 of [Appendix 8](#).

A10.1.2 Proposed Solution

See section A8.2 of [Appendix 8](#).

A10.1.3 System Architecture

The System will have five main components:

- Acquisitions Management Subsystem (Java or Delphi)
- Financial Management Subsystem (Java or Delphi)
- System Controls Subsystem
- Point of Sale Subsystem (POSS)
- Database Backbone (MySQL)

[Figure A10-1](#) provides the information topology chart (ITC) and [Figure A10-2](#) shows the object flow diagram (OFD). Notice that on the ITC there are additional entities added under the Controls Subsystem, compared to the diagram provided in [Appendix 9](#). This represents a refinement of the ITC provided in the requirements specification.

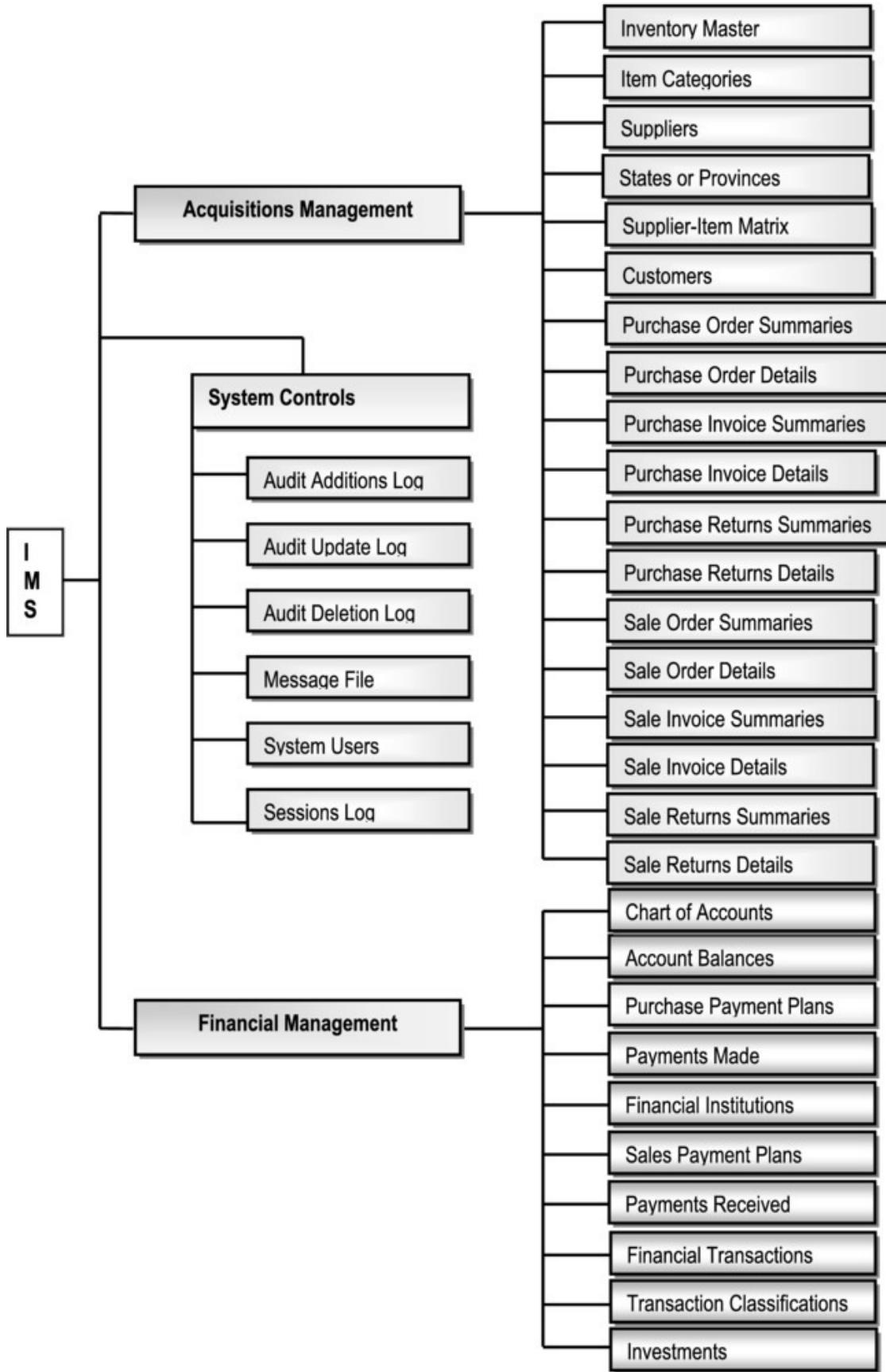


Figure A10-1. IMS Information Topology Chart

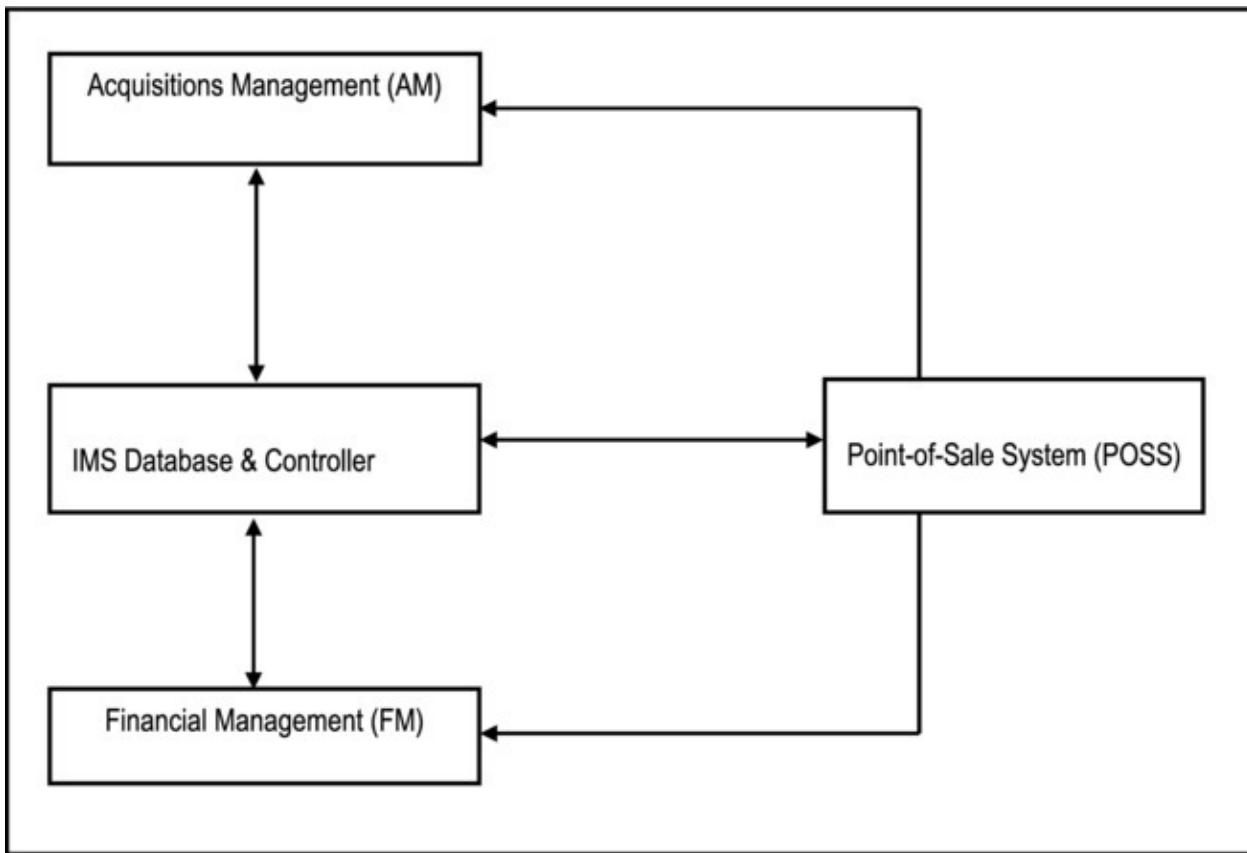


Figure A10-2. IMS Object Flow Diagram

A10.2 Database Specification

The database specification will proceed under the following captions:

- Introduction
- Acquisitions Management Subsystem
- Financial Management Subsystem
- System Controls Subsystem

A10.2.1 Introduction

The database specification of the system will be covered in this section. The methodology employed for database specification is the *object/entity specification grid* (O/ESG) developed by the current author. Following is a summary of the conventions used:

Each information entity referenced is identified by a reference code and a descriptive name.

- For each entity the attributes (data elements) to be stored are identified.
- The entities as presented, will easily transition into a set of normalized relations in a normalized relational database.
- Data elements that will be implemented as foreign keys, in the normalized relational database, are identified by comment in curly braces, specifying what entity they reference.
- For each attribute, the physical characteristics will be given (as described in the next section); the attributes implementation name will be indicated in square brackets; it will be indicated whether the attribute is a foreign key.
- Indexes (including primary key or candidate keys) to be defined on the entity are indicated.
- For each entity a comment describing the data to be stored is provided. Additionally, the entities implementation name is indicated in square brackets.
- Each operation defined on an entity will be given an implementation name, indicated in square brackets.

Naming of database objects will be very important for the following reasons:

- The database will host several objects. Without a proper naming convention, it will be extremely difficult to keep track of them.
- The naming convention will enable us to easily categorize database objects on sight.

[Figure A10-3](#) provides the object naming convention for the project.

<p>Object Name: SSXXXXXX_MMn where SS represents the system or subsystem abbreviation; MMn represents the object mode or purpose (1-3 bytes); XXXXXXXXX represents the descriptive name of the object (6-8 bytes).</p>
<p>Valid subsystem abbreviations include:</p> <p>AM: Acquisitions Management FM: Financial Management SC: System Controls</p>
<p>Valid mode abbreviations include:</p> <p>BR: A base relation (if relational DB model) OT: An object type (if OO DB model) LVn: A logical view (e.g. LV1, LV2, etc.) NXn: An index to a base table or object type (e.g. NX1, NX2, etc.) PK: Primary Key FKn: Foreign Key (e.g. FK1, FK2, etc.) ICn: Integrity Constraint (e.g. IC1, IC2, etc.) AO: An ADD operation MO: An MODIFY operation ZO: A DELETE (Zap) operation IO: An INQUIRE operation FO: A FORECAST operation. RO: A REPORT operation XO: A utility operation DS: A database synonym or alias of a known database table DC: A database constraint DT: A database Trigger DP: A database procedure or function DK: A database package MF: A Message file — a special purpose database table (file) to store the text (and other essential details) for diagnostic error and status messages</p>
<p>The descriptor used for a database base relation or object type is consistently used for other objects that directly relate to that object. For example, the objects used for the management of inventory items may be:</p> <ul style="list-style-type: none"> ▪ AMInvMaster_BR — a base relation to store data on inventory items ▪ AMInvMaster_NX1 — an index on the base relation ▪ AMInvMaster_AO — an operation to ADD inventory items ▪ AMInvMaster_MO — an operation to MODIFY inventory items ▪ AMInvMaster_ZO — an operation to DELETE inventory items ▪ AMInvMaster_IO — an operation to INQUIRE on inventory items ▪ AMInvMaster_RO — an operation to REPORT on inventory items ▪ AMInvMaster_XO — a utility operation related to inventory items ▪ AMInvMaster_LV1 — a logical view of the base relation
<p>Attribute implementation names are mere abbreviations of their more descriptive names.</p>

Figure A10-3. Object Naming Conventions

A10.2.2 Acquisitions Management Subsystem

Following is the O/ESG for each information entity comprising this subsystem (Figure A10-4):

<p>E1. Inventory Master [AMInvMaster_BR] consists of the following:</p>
<p>Attributes:</p> <ul style="list-style-type: none">1. Item Code [AMitmCd] A82. Item Name / Description [AMitmDes] [A30]3. Item Category [AMitmCat] {Refers to E2} [A4]4. Quantity on Hand [AMQty] [N(7,2)]5. Reorder Quantity [AMLowQty] [N(7,2)]6. Last Unit Price [AMLastPrice] [N(9,2)]7. Average Unit Price [AMAvgPrice] [N(9,2)]8. Last Selling Price [AMPrevPrice] [N(9,2)]9. Current Selling Price [AMPrice] [N(9,2)]10. Account Number [AMAcctNum] {Refers to E19} [N10]11. UPC Code [AMUPC] [N12]12. SKU Number [AMSKU] [A6]
<p>Comments:</p> <ul style="list-style-type: none">1. This entity stores information about inventory items2. Item Codes [1] will consist of a 4 digit alphabetic code combined with a 4 digit sequential number.
<p>Indexes:</p> <ul style="list-style-type: none">1. Primary Key: [1] (Constraint Name is AMInvMasterPK)2. AMInvMasterNX2 on [2]3. AMInvMasterNX3 on [3,1]4. AMInvMasterNX4 on [3,2]5. AMInvMasterNX5 on [11]6. AMInvMasterNX6 on [12]
<p>Valid Operations:</p> <ul style="list-style-type: none">1. Add Inventory Item [AMInvMaster_AO]2. Modify Inventory Item [AMInvMaster_MOO]3. Delete Inventory Item [AMInvMaster_ZO]4. Inquire Inventory Item [AMInvMaster_IO]5. Report Inventory Item [AMInvMaster_RO]

E2. Item Category [AMItemCat_BR] consists of the following:

Attributes:

1. Category Code [AMItemCat] [A4]
2. Category Description [AMCatDescr] [A30]

Comments:

1. This entity stores information about Item Categories
2. Category Code [1] will consist of a 4 digit alphabetic code combined with a 4 digit sequential number.

Indexes:

1. Primary Key: [1] (Constraint Name is **AMItemCatPK**)
2. AMItemCatNX2 on [2]

Valid Operations:

1. Add Item Category [AMItemCat_AO]
2. Modify Item Category [AMItemCat_MO]
3. Delete Item Category [AMItemCat_ZO]
4. Inquire Item Category [AMItemCat_IO]

E3. Supplier [AMSupplier_BR] consists of the following:

Attributes:

1. Supplier Code [AMSuppCode] [A8]
2. Supplier Name [AMSuppName] [A30]
3. A Address Line 1 [AMAddr1] [A30]
4. Address Line 2 [AMAddr2] [A30]
5. State or Province Code [AMState] {Refers to E4} [A4]
6. Zip Code [AMZip] [N8]
7. Telephone Number(s) [AMPhoneNo] [N10]
8. Fax Number [AMFaxNo] [N10]
9. Email Address [AMEmail] [A30]
10. Contact Person [AMContact] [A30]
11. Account Number [AMAccountNo] {Refers to E19} [N10]
12. Ordering Preference [AMOrderPref] [A30]

Comments:

1. This entity stores information about Suppliers
2. Supplier Code [1] will consist of a 4 digit alphabetic code combined with a 4 digit sequential number.

Indexes:

1. Primary Key: [1] (Constraint Name is **AMSupplierPK**)
2. AMSupplierNX2 on [2]
3. AMSupplierNX3 on [11]

Valid Operations:

1. Add Supplier [AMSupplier_AO]
2. Modify Supplier [AMSupplier_MO]
3. Delete Supplier [AMSupplier_ZO]
4. Inquire Supplier [AMSupplier_IO]
5. Report [AMSupplier_RO]

E4. State or Province [AMState_BR] consists of the following:

Attributes:

1. State / Province Code [AMStateCode] [A4]
2. State / Province Name [AMStateName] [A30]

Comments:

This entity stores the List of States and Provinces

Indexes:

1. Primary Key: [1] (Constraint Name is AMStatePK)

Valid Operations:

1. Add State or Province [AMState_AO]
2. Delete State or Province [AMState_ZO]
3. Inquire State or Province [AMState_IO]

E5. Supplier-Item Matrix [AMItemSupp_BR] consists of the following:

Attributes:

1. Supplier Code [AMSuppCode_BR] {Refers to E3} [A8]
2. Item Code [AMItemCode] {Refers to E1} [A8]

Comments:

This entity stores information about which items come from which Supplier

Indexes:

1. Primary Key: [1, 2] (Constraint Name is AMItemSuppPK)

Valid Operations:

1. Add Supplier-Item Matrix [AMItemSupp_AO]
2. Delete Supplier-Item Matrix [AMItemSupp_ZO]
3. Inquire Supplier-Item Matrix [AMItemSupp_IO]
4. Report Supplier-Item Matrix [AMItemSupp_RO]

E6. Customer [AMCustomer_BR] consists of the following:

Attributes:

1. Customer Code [AMCustCode] [A8]
2. Customer Name [AMCustName] [A30]
3. Address Line 1 [AMAddr1] [A30]
4. Address Line 2 [AMAddr2] [A30]
5. State / Province Code [AMState] {Refers to E4} [A4]
6. Zip Code [AMZip] [N8]
7. Telephone Number(s) [AMPhoneNo] [N10]
8. Fax Number (s) [AMFaxNo] [N10]
9. Email Address [AMEmail] [A30]
10. Contact Person [AMContact] [A30]
11. Account Number [AMAcctNo] {Refers to E19} [N20]
12. Billing Preferences [AMBillingPref] [A30]

Comments:

1. This entity stores information about Customers
2. Customer Code will consist of a 4 digit alphabetic code combined with a 4 digit sequential number.

Indexes:

1. Primary Key: [1] (Constraint Name is **AMCustomerPK**)
2. AMCustomerNX2 on [2]
3. AMCustomerNX3 on [11]

Valid Operations:

1. Add Customer [AMCustomer_AO]
2. Modify Customer [AMCustomer_MO]
3. Delete Customer [AMCustomer_ZO]
4. Inquire Customer [AMCustomer_IO]

E7. Purchase Order Summary [AMPOSum_BR] consists of the following:

Attributes:

1. Purchase Order Number [AMPONum] [N8]
2. Purchase Order Supplier Code [AMSuppCode] {Refers to E3} [N8]
3. Purchase Order Date [AMPODate] [N8]
4. Purchase Order Status [AMPOStatus] (Filled/ Partial/ Outstanding) [A30]
5. Purchase Order Estimated Amount [AMPOAmount] [N(11,2)]
6. Purchase Order Estimated Discount [AMPODiscount] [N(11,2)]

Comments:

1. This entity stores information about Purchase Orders
2. PO Code consists of 4 digit year concatenated with 4 digit sequential number (**YYYYSSSS**)

Indexes:

1. Primary Key: [1] (Constraint Name is **AMPOSumPK**)
2. AMPOSumNX2 on [2, 3]

Valid Operations:

1. Add Purchase Order [AMPOSum_AO]
2. Modify Purchase Order [AMPOSum_MO]
3. Delete Purchase Order [AMPOSum_ZO]
4. Inquire Purchase Order [AMPOSum_IO]
5. Report Purchase Order [AMPOSum_RO]

E8. Purchase Order Detail [AMPODet_BR] consists of the following:

Attributes:

1. Purchase Order Number [AMPONumber] {Refers to E7} [N8]
2. Item Code [AMitmCode] {Refers to E1} [N8]
3. Quantity Ordered [AMQty] [N(7,2)]
4. Order Unit Price [AMPrice] [N(9,2)]

Comments:

This entity stores information about Purchase Order Details

Indexes:

1. Primary Key: [1,2] (Constraint Name is AMPODetPK)

Valid Operations:

1. Add Purchase Order Detail [AMPODet_AO]
2. Modify Purchase Order Detail [AMPODet_MO]
3. Delete Purchase Order Detail [AMPODet_ZO]
4. Inquire Purchase Order Detail [AMPODet_IO]
5. Report Purchase Order Detail [AMPODet_RO]

E9. Purchase Invoice Summary [AMPurchInv_BR] consists of the following:

Attributes:

1. Purchase Invoice Identification Number [AMPIIDNum] [N10]
2. Invoice Number [AMInvNum] [A8]
3. Invoice Supplier Code [AMSuppCode] {Refers to E3} [A8]
4. Invoice Date [AMInvDate] [N8]
5. Related Purchase Order Number [AMPONumber] {Refers to E7} [N8]
6. Invoice Amount [AMInvAmount] [N(11,2)]
7. Invoice Amount Outstanding [AMAmtOwed] [N(11,2)]
8. Discount [AMDiscount] [N(11,2)]
9. Tax [AMTax] [N(9,2)]
10. Comment [AMComment] [A30]
11. Transaction Reference Number [AMTransRefNum] {Refers to E26} [A13]

Comments:

1. This entity stores information about Purchase Invoices
2. PI Identification Number consists of the 4 digit year, 2 digit month and 4 digit sequential number (YYYYMMSSSS)
3. Transaction Reference Number coding system consists of the literal "PI" concatenated with the PI Identification Number (PIYYYYMMSSSS)

Indexes:

1. Primary Key: [1] (Constraint Name is AMPurchInvPK)
2. AMPurchInvNX2 on [3, 2, 4]
3. AMPurchInvNX3 on [11]

Valid Operations:

1. Add Purchase Invoice [AMPurchInv_AO]
2. Modify Purchase Invoice [AMPurchInv_MO]
3. Delete Purchase Invoice [AMPurchInv_ZO]
4. Inquire Purchase Invoice [AMPurchInv_IO]
5. Report Purchase Invoice [AMPurchInv_RO]

E10. Purchase Invoice Detail [AMPIDet_BR] consists of the following:**Attributes:**

1. Purchase Invoice Identification [AMPIIDNum] {Refers to E9} [N10]
2. Item Number [AMItemNum] {Refers to E1} [A8]
3. Item Quantity [AMQty] [N(7,2)]
4. Item Unit Price [AMPrice] [N(9,2)]

Comments:

This entity stores information about Purchase Invoice Details

Indexes:

1. Primary Key: [1,2] (Constraint Name is **AMPIDetPK**)

Valid Operations:

1. Add Purchase Invoice Detail [AMPIDet_AO]
2. Modify Purchase Invoice Detail [AMEPIDet_MO]
3. Delete Purchase Invoice Detail [AMPIDet_ZO]
4. Inquire Purchase Invoice Detail [AMPIDet_IO]
5. Report Purchase Invoice Detail [AMPIDet_RO]

E11. Purchase Returns Summary [AMPRSum_BR] consists of the following:**Attributes:**

1. Purchase Invoice Identification [AMPIIDNum] {Refers to E9} [N10]
2. Return Date [AMRetDate] [N8]
3. Return Amount [RetAMAmount] [N(11,2)]
4. Purchase Return ID [AMReturnID] [N10]
5. Transaction Reference Number [AMTransRefNum] {Refers to E26} [A13]

Comments:

1. This entity stores information about Purchase Returns
2. PR ID consists of 4 digit year, 2 digit month and 4 digit sequential number (YYYYMMSSSS)
3. Transaction Reference Number coding system consists of the literal "PR" concatenated with the PR Identification Number (PRYYYYMMSSSS)

Indexes:

1. Primary Key: [4] (Constraint Name is **AMPRSumPK**)
2. AMPRSumNX2 on [1,2]
3. AMPRSumNX3 on [5]

Valid Operations:

1. Add Purchase Returns [AMPRSum_AO]
2. Delete Purchase Returns [AMPRSum_ZO]
3. Inquire Purchase Returns [AMPRSum_IO]
4. Report Purchase Returns [AMPRSum_RO]

E12. Purchase>Returns Detail [AMPRDetail_BR] consists of the following:

Attributes:

1. Purchase Return ID [AMReturnID] {Refers to E11} [N10]
2. Item Code [AMItemCode] {Refers to E1} [A8]
3. Quantity Returned [AMQtyRet] [N(7,2)]
4. Return Unit Price [AMRetPrice] [N(9,2)]
5. Comment [AMComment] [A45]

Comments:

This entity stores information about Purchase Return Details

Indexes:

1. Primary Key: [1,2] (Constraint Name is **AMPRDetPK**)

Valid Operations:

1. Add Purchase Returns Detail [AMPRDetail_AO]
2. Delete Purchase Returns Detail [AMPRDetail_ZO]
3. Inquire Purchase Returns Detail [AMPRDetail_IO]
4. Report Purchase Returns Detail [AMPRDetail_RO]

E13. Sale Order Summary [AMSOSum_BR] consists of the following:

Attributes:

1. Sales Order Identification Number [MSOIDNum] [N8]
2. Sales Order Number [MSONum] [A8]
3. Sales Order Customer Code [MSOCustCode] {Refers to E6} [A8]
4. Sales Order Date [MSODate] [N8]
5. Sales Order Status [MSOStatus] (Filled/ Partial/ Outstanding) [A1]
6. Sales Order Estimated Amount [MSOAmount] [N(11,2)]
7. Sales Order Estimated Discount [MSODiscount] [N(11,2)]

Comments:

1. This entity stores information about Sales Orders
2. Sale Order ID Coding system consists of 4 digit year concatenated with 4 digit sequential number (**YYYYSSSS**)

Indexes:

1. Primary Key: [1] (Constraint Name is **MSOSumPK**)
2. MSOSumNX2 on [3,2,4]

Valid Operations:

1. Add Sale Order [MSOSum_AO]
2. Modify Sale Order [MSOSum_MO]
3. Delete Sale Order [MSOSum_ZO]
4. Inquire Sale Order [MSOSum_IO]
5. Report Sale Order [MSOSum_RO]

E14. Sales Order Detail [AMSODetail_BR] consists of the following:

Attributes:

1. Sales Order Identification Number [AMSOIDNum] {Refers to E13} [N8]
2. Item Code [AMItemCode] {Refers to E1} [A8]
3. Quantity Ordered [AMQty] [N(7,2)]
4. Anticipated Unit Price [AMPrice] [N(9,2)]

Comments:

This entity stores information about Sales Order Details

Indexes:

1. Primary Key: [1,2] (Constraint Name is **AMSODetailPK**)

Valid Operations:

1. Add Sale Order Detail [AMSODetail_AO]
2. Modify Sale Order Detail [AMSODetail_MO]
3. Delete Sale Order Detail [AMSODetail_ZO]
4. Inquire Sale Order Detail [AMSODetail_IO]
5. Report Sale Order Detail [AMSODetail_RO]

E15. Sales Invoice Summary [AMSISSum_BR] consists of the following:

Attributes:

1. Sales Invoice Identification Number [AMSIIDNum] [N10]
2. Sales Invoice Number [AMSINum] [A8]
3. Sales Invoice Customer Code [AMSCustCode] {Refers to E6} [A8]
4. Sales Invoice Date [AMSIDate] [N8]
5. Related Sale Order ID [AMSOIDNum] {Refers to E13} [N8]
6. Sales Invoice Amount [AMSIAmount] [N(11,2)]
7. Sales Invoice Amount Outstanding [AMSIOWed] [N(11,2)]
8. Sales Invoice Discount [AMSIDiscount] [N(11,2)]
9. Sales Invoice Tax [AMSITax] [N(9,2)]
10. Transaction Reference Number [AMTransRefNum] {Refers to E26} [A13]
11. Sales Invoice Comment [AMComment] [A30]

Comments:

1. This entity stores information about Sales Invoices
2. Sales Invoice ID coding system consists of the 4 digit year, 2 digit month and 4 digit sequential number (**YYYYMMSSSS**)
3. Transaction Reference Number coding system consists of the literal "SI" concatenated with the SI Identification Number (**SIYYYYMMSSSS**)

Indexes:

1. Primary Key: [1] (Constraint Name is **AMSISSumPK**)
2. AMSISSumNX2 on [2,3,4]
3. AMSISSumNX3 on [10]

Valid Operations:

1. Add Sale Invoice [AMSISSum_AO]
2. Modify Sale Invoice [AMSISSum_MO]
3. Delete Sale Invoice [AMSISSum_ZO]
4. Inquire Sale Invoice [AMSISSum_IO]
5. Report Sale Invoice [AMSISSum_RO]

E16. Sales Invoice Detail [AMSIIDDetail_BR] consists of the following:

Attributes:

1. Sales Invoice Identification Number [**AMSIIDNum**] {Refers to E15} [N10]
2. Item Code [**AMItemCode**] {Refers to E1} [A8]
3. Item Quantity [**AMQty**] [N(7,2)]
4. Item Unit Price [**AMPPrice**] [N(9,2)]

Comments:

This entity stores information about Sales Invoice Details

Indexes:

1. Primary Key: [1,2] (Constraint Name is **AMSIIDDetailPK**)

Valid Operations:

1. Add Sale Invoice Detail [**AMSIIDDetail_AO**]
2. Modify Sale Invoice Detail [**AMSIIDDetail_MO**]
3. Delete Sale Invoice Detail [**AMSIIDDetail_ZO**]
4. Inquire Sale Invoice Detail [**AMSIIDDetail_IO**]
5. Report Sale Invoice Detail [**AMSIIDDetail_RO**]

E17. Sales Return Summary [AMSRSum_BR] consists of the following:

Attributes:

1. Sale Return ID [**AMRetID**] [N10]
2. Sales Invoice Identification Number [**AMSIIDNum**] {Refers to E15} [N10]
3. Return Date [**AMRetDate**] [N8]
4. Return Amount [**AMRetAmount**] [N(11,2)]
5. Transaction Reference Number [**AMTransRefNum**] {Refers to E26} [A13]

Comments:

1. This entity stores information about Sales Returns
2. Purchase Return ID Coding System consists of the 4 digit year, 2 digit month and 4 digit sequential number (YYYYMMSSSS)
3. Transaction Reference Number coding system consists of the literal "SR" concatenated with the SI Identification Number (SRYYYYMMSSSS)

Indexes:

1. Primary Key: [1] (Constraint Name is **AMSRSumPK**)
2. AMSRSumNX2 on [2,3]
3. AMSRSumNX3 on [5]

Valid Operations:

1. Add Sale Returns [**AMSRSum_AO**]
2. Modify Sale Returns [**AMSRSum_MO**]
3. Delete Sale Returns [**AMSRSum_ZO**]
4. Inquire Sale Returns [**AMSRSum_IO**]
5. Report Sale Returns [**AMSRSum_RO**]

E18. Sales Return Detail [AMSRDetail_BR] consists of the following:
Attributes:
1. Sale Return ID [AMRetID] {Refers to E17} [N10] 2. Item Code [AMItemCode] {Refers to E1} [A8] 3. Quantity Returned [AMQty] [N(7,2)] 4. Return Unit Price [AMPrice] [N(9,2)] 5. Comment [AMComment] [A30]
Comments:
This entity stores information about Sales Returns
Indexes:
1. Primary Key: [1,2] (Constraint Name is AMSRDetailPK)
Valid Operations:
1. Add Sale Return Detail [AMSRDetail_AO] 2. Delete Sale Return Detail [AMSRDetail_ZO] 3. Inquire Sale Return Detail [AMSRDetail_IO] 4. Report Sale Return Detail [AMSRDetail_RO]

Figure A10-4. O/ESG for the Acquisitions Management Subsystem

A10.2.3 Financial Management Subsystem

Following is the O/ESG for each information entity comprising this subsystem ([Figure A10-5](#)):

E19. Chart of Accounts [FMChrtAccts_BR] consists of the following:**Attributes:**

1. Account Number [FMAcctNum] [N10]
2. Account Description [FMAcctDesc] [A30]
3. Summary / Detail Flag [FMFlag] [A1]
4. Parent Account [FMParent] {Refers to E19} [N10]

Comments:

1. This entity stores information about the Various Accounts in the system
2. Account number coding system consists of 10 byte numeric code where the first 4 bytes represents the department and the last 6 bytes represent the account (**DDDDAAAAAA**)

Indexes:

1. Primary Key: [1] (Constraint Name is **FMChrtAcctsPK**)
2. FMChrtAcctsNX2 on [2]

Valid Operations:

1. Add Chart of Accounts [**FMChrtAccts_AO**]
2. Delete Chart of Accounts [**FMChrtAccts_ZO**]
3. Inquire Chart of Accounts [**FMChrtAccts_IO**]
4. Report Chart of Accounts [**FMChrtAccts_RO**]

E20. Account Balances Log [FMAcctBal_BR] consists of the following:**Attributes:**

1. Financial Period [FMPPeriod] [N6]
2. Account Number [FMAccountNum] {Refers to E19} [N10]
3. Balance [FMBalance] [N(11,2)]
4. Comment [FMComment] [A30]

Comments:

This entity stores information about the Account Balances

Indexes:

1. Primary Key: [1,2] (Constraint Name is **FMAcctBalPK**)

Valid Operations:

1. Add Account Balances [**FMAcctBal_AO**]
2. Modify Account Balances [**FMAcctBal_MO**]
3. Delete Account Balances [**FMAcctBal_ZO**]
4. Inquire Account Balances [**FMAcctBal_IO**]
5. Report Account Balances [**FMAcctBal_RO**]

E21. Purchase Invoice Payment Plan [FMPIPP_BR] consists of the following:**Attributes:**

1. Purchase Invoice Identification Number [FMPIIDNum] {Refers to E9} [N10]
2. Required Day of Month [FMDay] [N2]
3. Number of Payments Required [FMNumPaymts] [N3]
4. Payment Begin Date [FMDate] [N8]

Comments:

This entity stores information about Invoice Payment Plans

Indexes:

1. Primary Key: [1] (Constraint Name is FMPIPPK)

Valid Operations:

1. Add Purchase Invoice Payment Plan [FMPIPP_AO]
2. Modify Purchase Invoice Payment Plan [FMPIPP_MO]
3. Delete Purchase Invoice Payment Plan [FMPIPP_ZO]
4. Inquire Purchase Invoice Payment Plan [FMPIPP_IO]
5. Report Purchase Invoice Payment Plan [FMPIPP_RO]

E22. Payments Made Log [FMPaymtsMade_BR] consists of the following:**Attributes:**

1. Payment Identification Code [FMPayIDCode] [N10]
2. Purchase Invoice Identification Number [FMPIIDNum] {Refers to E9} [N10]
3. Payment Date [FMPayDate] [N8]
4. Amount Paid [FMAmtPaid] [N(11,2)]
5. Payment Type [FMPayType] (Cash / Check / Credit Card) [A4]
6. Institution Code [FMInstCode] {Refers to E23} [N8]
7. Check / Credit Account Number [FMAcctNum] [N16]
8. Transaction Reference Number [FMTransRefNum] {Refers to E26} [A13]
9. Comment [FMComment] A30

Comments:

1. This entity stores information about Payments Made
2. Purchase Payment ID Coding System consists of the 4 digit year, 2 digit month and 4 digit sequential number (YYYYMMSSSS)
3. Transaction Reference Number coding system consists of the literals "PP1" and "PP2" concatenated with the Purchase Payment Identification Number (PP1YYYYMMSSSS) or (PP2YYYYMMSSSS)

Indexes:

1. Primary Key: [1] (Constraint Name is FMPaymtsMadePK)
2. FMPaymtsMadeNX2 on [2,3]
3. FMPaymtsMadeNX3 on [8]

Valid Operations:

1. Add Payments Made [FMPaymtsMade_AO]
2. Delete Payments Made [FMPaymtsMade_ZO]
3. Inquire Payments Made [FMPaymtsMade_IO]
4. Report Payments Made [FMPaymtsMade_RO]

E23. Financial Institution [FMFinInst_BR] consists of the following:

Attributes:

1. Institution Code [FMIInstCode] [A8]
2. Institution Name [FMIInstName] [A30]
3. Address Line 1 [FMAAddr1] [A30]
4. Address Line 2 [FMAAddr2] [A30]
5. State / Province Code [FMState] {Refers to E4} [A4]
6. Zip Code [FMZip] [N8]
7. Telephone Number(s) [FMPhoneNo] [N10]
8. Fax Number(s) [FMFaxNo] [N10]
9. Contact Person [FMContact] [A30]
10. Email Address [FMEmail] [A30]

Comments:

1. This entity stores information about Financial Institutions
2. Institution Code will consist of a 4 digit alphabetic code combined with a 4 digit sequential number.

Indexes:

1. Primary Key: [1] (Constraint Name is **FMFinInstPK**)

Valid Operations:

1. Add Financial Institutions [FMFinInst_AO]
2. Delete Financial Institutions [FMFinInst_ZO]
3. Inquire Financial Institutions [FMFinInst_IO]
4. Report Financial Institutions [FMFinInst_RO]

E24. Sales Invoice Payment Plan [FMSIPP_BR] consists of the following:

Attributes:

1. Sales Invoice Identification Number [FMSIIDNum] {Refers to E15} [N10]
2. Required Day of Month [FMDay] [N2]
3. Number of Payments Required [FMNumPaymts] [N3]
4. Payment Begin Date [FMDate] [N8]

Comments:

This entity stores information about Sales Invoices Payment Plans

Indexes:

1. Primary Key: [1] (Constraint Name is **FMSIPPPK**)

Valid Operations:

1. Add Sales Invoice Payment Plan [FMSIPP_AO]
2. Delete Sales Invoice Payment Plan [FMSIPP_ZO]
3. Inquire Sales Invoice Payment Plan [FMSIPP_IO]
4. Report Sales Invoice Payment Plan [FMSIPP_RO]

E25. Payments Received [FMPayRecv_BR] consists of the following:

Attributes:

1. Payment Identification Code [FMPayIDCode] [N10]
2. Sales Invoice Identification Number [FMSIIDNum] {Refers to E15} [N10]
3. Payment Received Date [FMDate] [N8]
4. Amount Received [FMAmount] [N(11,2)]
5. Payment Type [FMPayType] (Cash / Check / Credit Card) [A4]
6. Institution Code [FMInstCode] {Refers to E23} [N8]
7. Check / Credit Account Number [FMAcctNum] [N16]
8. Transaction Reference Number [FMTransRefNum] {Refers to E26} [A13]
9. Comment [FMComment] [A30]

Comments:

1. This entity stores information about Payments Received
2. Sale Payment ID coding system consists of the 4 digit year, 2 digit month and 4 digit sequential number (YYYYMMSSSS)
3. Transaction Reference Number coding system consists of the literals "SP1" and "SP2" concatenated with the Sale Payment Identification Number (SP1YYYYMMSSSS) or (SP2YYYYMMSSSS)

Indexes:

1. Primary Key: [1] (Constraint Name is FMPayRecvPK)
2. FMPayRecvNX2 on [2, 3]
3. FMPayRecvNX3 on [8]

Valid Operations:

1. Add Payments Received [FMPayRecv_AO]
2. Delete Payments Received [FMPayRecv_ZO]
3. Inquire Payments Received [FMPayRecv_IO]
4. Report Payments Received [FMPayRecv_RO]

E26. Financial Transactions [FMFinanTrans_BR] consists of the following:

Attributes:

1. Transaction Date [FMDDate] [N8]
2. Transaction Classification Code [FMClassCode] {Refers to E27} [A3]
3. Account Number [FMAcctNum] {Refers to E18} [N10]
4. Debit Amount [FMDdebit] [N(11,2)]
5. Credit Amount [FMCredit] [N(11,2)]
6. Accounting Period [FMPPeriod] [N6]
7. Transaction Reference Number [FMTransRefNum] (see Note below) [A13]

Comments:

1. This entity stores information about Financial Transactions
2. Transaction Reference coding system: CCCYYYYMMSSSS where CCC represents the classification code

Indexes:

1. Primary Key: [7] (Constraint Name is FMFinanTransPK)

Valid Operations:

1. Add Financial Transactions [FMFinanTrans_AO]
2. Modify Financial Transactions [FMFinanTrans_MO]
3. Delete Financial Transactions [FMFinanTrans_ZO]
4. Inquire Financial Transactions [FMFinanTrans_IO]
5. Report Financial Transactions [FMFinanTrans_RO]

Note: Based on the classification code, Transaction Reference ties back to one of the following entities:

PI: Purchase Invoice Transactions (**E9**)

PR: Purchase Return Transactions (**E11**)

SI: Sales Invoice Transactions (**E15**)

SR: Sale Return Transactions (**E17**)

PP1: Cash Purchase Payments Transactions (**E22**)

PP2: Credit Purchase Payments Transactions (**E22**)

SP1: Cash Sales Payments Transactions (**E25**)

SP2: Credit Sales Payments Transactions (**E25**)

INV: Investment Transactions (**E28**)

Comments:

These will be implemented via logical views.

E27. Transaction Classification [FMTranClas_BR] consists of the following:

Attributes:

1. Transaction Classification Code [**FMTransCode**] [A3]
2. Transaction Classification Description [**FMTransDesc**] [A30]

Comments:

This entity stores information about Transaction Classifications

Indexes:

1. Primary Key: [1] (Constraint Name is **FMTransClasPK**)

Valid Operations:

1. Add Transaction Classifications [**FMTranClas_AO**]
2. Delete Transaction Classifications [**FMTranClas_ZO**]
3. Inquire Transaction Classifications [**FMTranClas_IO**]
4. Report Transaction Classifications [**FMTranClas_RO**]

Note: Transaction Classifications include:

PI – Purchase Invoices

PR – Purchase Returns

SI – Sales Invoices

SR – Sales Returns

PP1 – Cash Purchase Payments

PP2 – Credit Purchase Payments

SP1 – Cash Sales Payments

SP2 – Credit Sales Payments

JED – Journal Entries Debit

JEC – Journal Entries Credit

INV – Investments

E28. Investments Log [FMIInvest_BR] consists of the following:
Attributes:
<ol style="list-style-type: none"> 1. Investment ID Code [FMIInvIDCode] [N10] 2. Investment Transaction Date [FMDDate] [N(11,2)] 3. Investment Amount [FMAmount] [N6] 4. Institution Code [FMInstCode] {Refers to E23} [N8] 5. Institutional Account Number [FMAcctNum] [N16] 6. Internal Account Number [FMIInternalRef] {Refers to E18} [N10] 7. Transaction Reference Number [FMTransRefNum] {Refers to E26} [A13] 8. Comment [FMComment] [A30]
Comments:
<ol style="list-style-type: none"> 1. This entity stores information about Investments 2. Investment ID coding system: YYYYMMSSSS 3. Transaction Reference coding system consists of the literals "INV" concatenated with the Investment Identification Number (INVYYYYMMSSSS)
Indexes:
<ol style="list-style-type: none"> 1. Primary Key: [1] (Constraint Name is FMIInvestPK) 2. FMIInvestNX2 on [2,4] 3. FMIInvestNX4 on [7]
Valid Operations:
<ol style="list-style-type: none"> 1. Add Investment [FMIInvest_AO] 2. Modify Investment [FMIInvest_MO] 3. Delete Investment [FMIInvest_ZO] 4. Inquire Investment [FMIInvest_IO] 5. Report Investment [FMIInvest_RO]

Figure A10-5. O/ESG for the Financial Management Subsystem

A10.2.4 Systems Control Subsystem

Following is the O/ESG for each information entity comprising this subsystem ([Figure A10-6](#)):

E29. Audit File for Additions [SCAudAdd_BR] includes the following:**Attributes:**

1. Session ID [SCAddID] [N12] {Refers to E34.1}
2. Add Code [SCAddCode] [N4]
3. Add Detail 1 [SCAddDet1] [A75]
4. Add Detail 2 [SCAddDet2] [A75]
5. Add Detail 3 [SCAddDet3] [A75]
6. Add Detail 4 [SCAddDet4] [A75]
7. Add File [SCAddFile] [A12]

Comments:

1. This table logs all additions of data to the system.
2. Session ID identifies session and will be in the form of a date and a sequence number (YYYYMMDDSSSS).
3. Add Code will be in the form of a sequence number.

Indexes:

1. Primary Key: [1,2] (Constraint Name is SCAudAddPK)

Valid Operations:

1. Add Audit File Add [SCAudAdd_AO]
2. Delete Audit File Add [SCAudAdd_ZO]
3. Inquire on Audit File Add [SCAudAdd_IO]
4. Report Audit File Add [SCAudAdd_RO]

E30. Audit File for Updates [SCAudUpd_BR] includes the following:**Attributes:**

1. Session ID [SCUpdateID] [N12] {Refers to E34.1}
2. Update Code [SCUpdateCode] [N4]
3. Before Detail 1 [SCUpdateBefDet1] [A75]
4. Before Detail 2 [SCUpdateBefDet2] [A75]
5. Before Detail 3 [SCUpdateBefDet3] [A75]
6. Before Detail 4 [SCUpdateBefDet4] [A75]
7. After Detail 1 [SCUpdateAftDet1] [A75]
8. After Detail 2 [SCUpdateAftDet2] [A75]
9. After Detail 3 [SCUpdateAftDet3] [A75]
10. After Detail 4 [SCUpdateAftDet4] [A75]
11. Update File [SCUpdateFile] [A12]

Comments:

1. This table logs all update of data in the system.
2. Session ID identifies session and will be in the form of a date and a sequence number (YYYYMMDDSSSS).
3. Update Code be in the form of a sequence number.

Indexes:

1. Primary Key: [1,2] (Constraint Name is SCAudUpdPK)

Valid Operations:

1. Add Audit File Update [SCAudUpd_AO]
2. Delete Audit File Update [SCAudUpd_ZO]
3. Inquire on Audit File Update [SCAudUpd_IO]
4. Report Audit File Update [SCAudUpd_RO]

E31. Audit File for Deletions [SCAudDel_BR] includes the following**Attributes:**

1. Session ID [SCDelID] [N12] {Refers to E34.1}
2. Delete Code [SCDelCode] [N4]
3. Delete Detail 1 [SCDelDetail1] [A75]
4. Delete Detail 2 [SCDelDetail2] [A75]
5. Delete Detail 3 [SCDelDetail3] [A75]
6. Delete Detail 4 [SCDelDetail4] [A75]
7. Delete Comment [SCDelComment] [A75]
8. Delete File [SCDelFile] [A12]

Comments:

1. This table logs all deletion of data from the system.
2. Session ID identifies session and will be in the form of a date and a sequence number (YYYYMMDDSSSS).
3. Delete Code will be in the form of a sequence number.

Indexes:

1. Primary Key: [1,2] (Constraint Name is SCAudDelPK)

Valid Operations:

1. Add Audit File Delete [SCAudDel_AO]
2. Delete Audit File Delete [SCAudDel_ZO]
3. Inquire on Audit File Delete [SCAudDel_IO]
4. Report Audit File Delete [SCAudDel_RO]

E32. Message File [SCMess_MF] includes the following:**Attributes:**

1. Message ID [SCMessID] [A7]
2. Message Description [SCMessDesc] [A75]

Comments:

1. This table defines system messages.
2. Message ID will be of the form SSEEEnn [SS = System or subsystem (AM, FM, SC), EEE = Entity Reference (E01...E31), nn = Sequence #].

Indexes:

1. Primary Key: [1] (Constraint Name is SCMessPK)

Valid Operations:

1. Add Message [SCMess_AO]
2. Delete Message [SCMess_ZO]
3. Retrieve Message [SCMess_RO]

E33. System Users [SCSysUser_BR] includes the following**Attributes:**

1. User Code [SCUserCode] [N7]
2. User Full Name [SCUserName] [A35]
3. User Short/Login Name [SCUserLogin] [A15]
4. User Password [SCUserPass] [A10]

Comments:

1. This table defines users who will be directly using the system..
2. The User Code will be of the form YYYYSSS (Y = Year, S = Sequence #).

Indexes:

1. Primary Key: [1] (Constraint Name is **SCSysUserPK**)
2. SCSysUserNX2 on [2]
3. SCSysUserNX3 on [3]

Valid Operations:

1. Add System User [**SCSysUser_AO**]
2. Modify System User [**SCSysUser_MO**]
3. Delete System User [**SCSysUser_ZO**]
4. Inquire on System User [**SCSysUser_IO**]
5. Report System User [**SCSysUser_RO**]

E34. Sessions Log [SCSessLog_BR] includes the following:**Attributes:**

1. Session ID [SCSessLogID] [N12]
2. User Code [SCSessLogUser] [N7] {Refers to E33.1}
3. Login Time [SCSessLoginTime] [N6]
4. Logout Time [SCSessLogoutTime] [N6]
5. Normal/Abnormal Flag [SCSessLogNormFlag] [A1]

Comments:

1. This table defines sessions of users using the system.
2. Session ID identifies session and will be in the form of a date and a sequence number (YYYYMMDDSSSS).
3. The Normal/Abnormal Flag indicates whether the session ended normally or abnormally (i.e. N for Normal or A for Abnormal).

Indexes:

1. Primary Key: [1] (Constraint Name is **SCSessLogPK**)

Valid Operations:

1. Add Session [**SCSessLog_AO**]
2. Delete Session [**SCSessLog_ZO**]
3. Inquire on Session [**SCSessLog_IO**]
4. Report Session [**SCSessLog_RO**]

E35. Session Mark [SCSessMark_BR] includes the following
Attributes:
1. Session Mark[SCSessionMark] [N4] 2. Session Date [SCSessionDate] [N8]
Comments:
This table is used to facilitate auditing by session. Each day, the Session Mark is initialized before users can access the system.
Indexes:
1. Primary Key: [2] (Constraint Name is SCSessMarkPK)
Valid Operations:
1. Initialize System [SCInit_XO]

Figure A10-6. O/ESG for the System Controls Subsystem

A10.3 Operations Specification

In this section, operation specifications for selected operations in the system are provided. The algorithms needed for some of the operations (for different entities) are similar. Therefore, in the interest of brevity, instead of repeating the same pseudo code for these similar operations, the following generic operation outlines will be referenced. The section proceeds as follows:

- Generic Pseudo-codes
- Acquisitions Management Subsystem
- Financial Management Subsystem
- System Controls Subsystem

A10.3.1 Generic Pseudo-codes

Generic pseudo-codes are provided for the ADD operation, the MODIFY operation and the DELETE operation in [Figure A10-7](#), [A10-8](#) and [A10-9](#) respectively.

```

START
WHILE (User wishes to continue)
    Accept Key Field(s);
    Check Record Absence or Existence in the primary file;
    IF      (Record Absent)
        Accept Non-key Fields;
        Validate Non-key Fields based on Validation Rules;
        WHILE(Any Error Exists),
            Re-display Non-key Fields for possible Update;
            Display appropriate error message(s);
            Validate Non-key Fields based on Validation Rules;
        END-WHILE;
        Re-display full Record for confirmation;
        IF      (Confirmation Obtained)
            Write New Record to the primary file;
            Write New Record to file SCAudAdd_BR via operation SCAudAdd_AO;
        ENDIF;
        ELSE Inform the User that nothing was saved; END-ELSE;
    ENDIF;
    ELSE Display Message ('Record already exists'); END-ELSE;
    Check if User wishes to quit and set an exit flag if necessary;
END WHILE;
Generate Edit-List;
STOP

```

Figure A10-7. Generic ADD Pseudo-code

```

START
WHILE (User wishes to continue)
    Accept Key Field(s);
    Check Record Absence or Existence in the primary file;
    IF      (Record Present)
        Retrieve Record and update Audit Log Fields (with before-values);
        Display Non-key Fields for possible Update;
        Validate Non-key Fields based on Validation Rules;
        WHILE(Any Error Exists),
            Re-display Non-key Fields for possible Update;
            Display appropriate error message(s);
            Validate Non-key Fields based on Validation Rules;
        END-WHILE;
        Re-display full Record for confirmation;
        IF      (Confirmation Obtained)
            Update Audit Log Fields (with current-values);
            Write New Record to file SCAudUpd_BR via operation SCAudUpd_AO;
            Update Record in the primary file;
        ENDIF;
        ELSE Inform the User that nothing was saved; END-ELSE;
    ENDIF;
    ELSE Display Message ('Record does not exists'); END-ELSE;
    Check if User wishes to quit and set an exit flag if necessary;
END-WHILE;
Generate Edit-List;
STOP

```

Figure A10-8. Generic MODIFY Pseudo-code

```

START
WHILE (User wishes to continue)
    Accept Key Field(s);
    Check Record Absence or Existence in the primary file;
    IF      (Record Present)
        Retrieve Record;
        Display full Record for confirmation;
        IF      (Deletion Confirmation Obtained)
            Update Audit Log Fields (with current-values);
            Write New Record to file SCAudDel_BR via operation SCAudDel_AO;
            Delete Record from the primary file;
        ENDIF;
        ELSE Inform the User that nothing was saved; END-ELSE;
    ENDIF;
    ELSE Display Message ('Record does not exists'); END-ELSE;
    Check if User wishes to quit and set an exit flag if necessary;
END WHILE;
Generate Edit-List;
STOP

```

Figure A10-9. Generic DELETE Pseudo-code

A10.3.2 Acquisitions Management Subsystem

The operation specifications for operations in the Acquisitions Management Subsystem (AMS) follow:

E1_A: Add Inventory Items

Operation Biography:

System: IMS
Subsystem: Acquisitions Management
Operation Name: **AMInvMaster_AO**
Operation Description: Facilitates addition of items to the Item Master table.
Operation Category: Mandatory
Complexity Rank: 8 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

New Item Form
AMInvMaster_BR — Inventory Master File (E1)
AMItemCat_BR — Item Categories (E2)
FMChrtAccts_BR — Chart of Accounts (E19)

Outputs:

AMInvMaster_BR — Inventory Master (E1)

Validation Rules:

1. Item Code must not previously exist
2. Category Code must already exist in **AMItemCat_BR**
3. Blank Item Name not allowed
4. Account Number must already exist in **FMChartAccts_BR**

Special Notes:

When a new Item is added or purchased, the Quantity on Hand and the Quantity Owned are adjusted.

Operation Outline: See Generic ADD pseudo-code.

E1_M: Modify Inventory Items

Operation Biography:

System: IMS
Subsystem: Acquisitions Management
Operation Name: **AMInvMaster_MO**
Operation Description: Facilitates modification of records in the Item Master table.
Operation Category: Important
Complexity Rank: 7 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

AMInvMaster_BR — Inventory Master File (E1)
AMIItemCat_BR — Item Categories (E2)
FMChrtAccts_BR — Chart of Accounts (E19)

Outputs:

AMInvMaster_BR — Inventory Master File (E1)

Validation Rules:

1. Item Code must previously exist
2. Category Code must already exist in **AMIItemCat_BR**
3. Blank Item Title not allowed
4. Account Number must already exist in **FMChrtAccts_BR**

Special Notes: None

Operation Outline: See Generic MODIFY pseudo-code

E1_Z: Delete Inventory Items

Operation Biography:

System: IMS
Subsystem: Acquisitions Management
Operation Name: **AMInvMaster_ZO**
Operation Description: Facilitates deletion of records from the Item Master table.
Operation Category: Mandatory
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

AMInvMaster_BR — Inventory Master File (E1)
AMItemCat_BR — Item Categories (E2)
FMChrtAccts_BR — Chart of Accounts (E19)

Outputs:

AMInvMaster_BR — Inventory Master File (E1)

Validation Rules:

Item Code must previously exist

Special Notes: None

Operation Outline: See Generic DELETE Pseudo-code

E1_I / E1_R: Inquiry/Report on Inventory Items

Operation Biography:

System: IMS
Subsystem: Acquisitions Management
Operation Name: AMInvMaster_IO / AMInvMaster_RO
Operation Description: Facilitates inquiry/report on Inventory Items.
Operation Category: Mandatory
Complexity Rank: 10 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

AMInvMaster_BR — Inventory Master File (E1)
AMItemCat_BR — Item Categories (E2)
FMChrtAccts_BR — Chart of Accounts (E19)

Outputs: Monitor / Printer

Validation Rules: None

Special Notes:

1. It will be possible to query Items via any of the following access paths:
 - 1.1 By Category and Item Name
 - 1.2 By Item Category & Code
 - 1.3 By Account Number
 - 1.4 UPC Code or SKU number
2. Each option will invoke one of four sub-operations (AMInvMaster_I1, AMInvMaster_I2, ... AMInvMaster_I4).
3. Utilizes the logical view AMInvMaster_LV1, which joins AMInvMaster_BR with AMItemCat_BR and FMChrtAccts_BR.

Operation Outline:

START: /* Inquire */

 While User Wishes to Continue

 Present the User with the options mentioned above;

 Depending on the User's choice, Invoke one of the sub-operations;

 End-While;

STOP.

Outline for AMInvMaster_I1:

START

 While User Wishes to Continue

 Prompt user for Item Name;

 Starting at that point in AMInvMaster_LV1, Load a Virtual Data Collection Object with all records
 until End-of-File;

 Display the Virtual Data Collection Object;

 End-While;

STOP.

{The rest of the operations will be similar}

Operations Specification continues ...

A10.3.3 Financial Management Subsystem

The operation specifications for operations in the Financial Management Subsystem (FMS) follow:

E19_A: Add Accounts to Chart of Accounts

Operation Biography:

System: IMS
Subsystem: Financial Management
Operation Name: **FMChrtAccts_AO**
Operation Description: Facilitates addition of records to the Chart of Accounts.
Operation Category: Mandatory
Complexity Rank: 8 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

New Chart of Accounts Form
FMCHrtAccts_BR — Chart of Accounts (E19)

Outputs:

FMCHrtAccts_BR — Chart of Accounts (E19)

Validation Rules:

1. The Account Number specified must not previously exist.

Special Notes: None

Operation Outline: See Generic ADD pseudo-code.

E19_M: Modify Accounts

Operation Biography:

System: IMS
Subsystem: Financial Management
Operation Name: **FMChrtAccts_AO**
Operation Description: Facilitates modification of Chart of Accounts records.
Operation Category: Mandatory
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs: FMCHrtAccts_BR — Chart of Accounts (E19)

Outputs: FMCHrtAccts_BR — Chart of Accounts (E19)

Validation Rules: Account Number specified must previously exist Chart of Accounts.

Special Notes: None

Operation Outline: See Generic MODIFY pseudo-code.

E19_Z: Delete Accounts

Operation Biography:

System: IMS
Subsystem: Financial Management
Operation Name: **FMChrtAccts_ZO**
Operation Description: Facilitates deletion of records from the Chart of Accounts.
Operation Category: Mandatory
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs: FMCHrtAccts_BR — Chart of Accounts (E19)

Outputs: FMCHrtAccts_BR — Chart of Accounts (E19)

Validation Rules: Account Number specified must previously exist Chart of Accounts.

Special Notes: None

Operation Outline: See Generic DELETE pseudo-code.

E19_I / E19_R: Inquiry/Report on Chart of Accounts

Operation Biography:

System: IMS
Subsystem: Financial Management
Operation Name: **FMChrtAccts_IO / FMChrtAccts_RO**
Operation Description: Facilitates inquiry/report on Chart of Accounts.
Operation Category: Important
Complexity Rank: 7 of 10
S Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

FMCHrtAccts_BR – Chart of Accounts (E19)

Outputs:

Monitor / Printer

Validation Rules: None

Special Notes: It will be possible to query Chart of Accounts by Account Number or Account Name.

Operation Outline:

START:

 While User Wishes to Continue
 Prompt user for Account Number or Account Name;
 Prompt user for preference (Accounts by Account Number or Account Name);
 If By Code
 Starting at that point in **FMCHrtAccts_BR** Load a Virtual Data
 Collection Object with all records until End-of-File;
 Display the Virtual Data Collection Object;
 End-If;
 If By Description
 Starting at that point in **FMCHrtAccts_BR**, Load a Virtual Data
 Collection Object with all records until End-of-File;
 Display the Virtual Data Collection Object;
 End-If;
 End-While;

STOP.

Operations Specification continues ...

A10.3.4 System Controls Subsystem

The operation specifications for operations in the System Controls Subsystem (SCS) follow:

E29_A: Add to the Audit Table for Data Additions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudAdd_AO
Operation Description: Facilitates addition of records to the Audit Table for Data Additions.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudAdd_BR — Audit Table for Data Additions

Outputs:

SCAudAdd_BR — Audit Table for Data Additions

Validation Rules: None

Special Notes:

1. This operation will be called with arguments for the attributes (columns) of SCAudAdd_BR.
2. The source table is the table written to by the operation that called this operation.

Operation Outline:

START

 Use the arguments received to load SCAudAdd_BR record fields;

 Write the new record to SCAudAdd_BR;

STOP.

■ **Note** This operation requires a different logic from the generic ADD operation.

E29_Z: Delete from the Audit Table for Data Additions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudAdd_ZO
Operation Description: Facilitates deletion of records from the Audit Table for Data Additions.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudAdd_BR — Audit Table for Data Additions

Outputs:

SCAudAdd_BR — Audit Table for Data Additions

Validation Rules: None

Special Notes:

The user will specify a starting Session ID, and an ending Session ID. Records between both sessions (inclusive) will be deleted.

Operation Outline:

START:

Prompt user for Starting Session and Ending Session;
Delete all records from SCAudAdd_BR that satisfy the condition
(Start Session <= SCAddID <= Stop Session);

STOP

■ **Note** This operation requires a different logic from the generic DELETE operation.

E29_I / E29_R: Inquiry/Report on the Audit Table for Data Additions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudAdd_IO / SCAudAdd_RO
Operation Description: Facilitates inquiry/report on the Audit Table for Data Additions.
Operation Category: Enhancement
Complexity Rank: 8 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudAdd_BR — Audit Table for Data Additions
SCSessLog_BR — Sessions Log

Outputs:

Monitor / Printer

Validation Rules: None

Special Notes:

1. It will be possible to query audited data additions By Session ID.
2. Utilizes the logical view SCAudAdd_LV which joins SCAudAdd_BR with SCSessLog_BR, and SCSSysUser_BR.

Operation Outline:

START: /* Inquire */

```
    While User Wishes to Continue
        Prompt user for Session ID;
        Starting at that point in SCAudAdd_LV, Load a Virtual Data Collection Object with all
            records until End-of-File;
        Display the Virtual Data Collection Object;
    End-While;
STOP.
```

E30_A: Add to Audit Table for Data Modifications

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: **SCAudUpd_AO**
Operation Description: Facilitates addition of records to the Audit Table for Data Modifications.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudUpd_BR — Audit Table for Data Modifications

Outputs:

SCAudUpd_BR — Audit Table for Data Modifications

Validation Rules: None

Special Notes:

1. This operation will be called with arguments for the attributes (columns) of **SCAudUpd_BR**.
2. The source table is the table written to by the operation that called this operation.

Operation Outline:

START

 Use the arguments received to load **SCAudUpd_BR** record fields;

 Write the new record to **SCAudUpd_BR**;

STOP.

E30_Z: Delete from Audit Table for Data Modifications

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudUpd_ZO
Operation Description: Facilitates deletion of records from the Audit Table for Data Modifications.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudUpd_BR — Audit Table for Data Modifications

Outputs:

SCAudUpd_BR — Audit Table for Data Modifications

Validation Rules: None

Special Notes:

The user will specify a starting Session ID, and an ending Session ID. Records between both sessions (inclusive) will be deleted.

Operation Outline:**START:**

Prompt user for Starting Session and Ending Session;
Delete all records from SCAudUpd_BR that satisfy the condition
(Start Session <= SCUpdID <= Stop Session);

STOP

E30_I / E30_R: Inquiry/Report on Audit Table for Data Modifications

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudUpd_IO / SCAudUpd_RO
Operation Description: Facilitates inquiry/report on the Audit Table for Data Modifications.
Operation Category: Enhancement
Complexity Rank: 8 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudUpd_BR — Audit Table for Data Modifications
SCSessLog_BR — Sessions Log

Outputs:

Monitor / Printer

Validation Rules: None

Special Notes:

1. It will be possible to query audited data modifications By Session ID.
2. Utilizes the logical view SCAudUpd_LV which joins SCAudUpd_BR with SCSessLog_BR, and SCSSysUser_BR.

Operation Outline:

START: /* Inquire */

 While User Wishes to Continue

 Prompt user for Session ID;

 Starting at that point in SCAudUpd_LV, Load a Virtual Data Collection Object with all
 records until End-of-File;

 Display the Virtual Data Collection Object;

 End-While;

STOP.

E31_A: Add to Audit Table for Data Deletions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudDel_AO
Operation Description: Facilitates addition of records to the Audit Table for Data Deletions.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudDel_BR — Audit Table for Data Deletions

Outputs:

SCAudDel_BR — Audit Table for Data Deletions

Validation Rules: None

Special Notes:

1. This operation will be called with arguments for the attributes (columns) of SCAudDel_BR.
2. The source table is the table written to by the operation that called this operation.

Operation Outline:

START

 Use the arguments received to load SCAudDel_BR record fields;

 Write the new record to SCAudDel_BR;

STOP.

E31_Z: Delete from Audit Table for Data Deletions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudDel_ZO
Operation Description: Facilitates deletion of records from the Audit Table for Data Deletions.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudDel_BR — Audit Table for Data Deletions

Outputs:

SCAudDel_BR — Audit Table for Data Deletions

Validation Rules: None

Special Notes:

The user will specify a starting Session ID, and an ending Session ID. Records between both sessions (inclusive) will be deleted.

Operation Outline:**START:**

Prompt user for Starting Session and Ending Session;
Delete all records from SCAudDel_BR that satisfy the condition
(Start Session <= SCDelID <= Stop Session);

STOP

E31_I / E31_R: Inquiry/Report on Audit Table for Data Deletions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCAudDel_IO / SCAudDel_RO
Operation Description: Facilitates inquiry/report on the Audit Table for Data Deletions.
Operation Category: Enhancement
Complexity Rank: 8 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCAudDel_BR — Audit Table for Data Deletions
SCSessLog_BR — Sessions Log

Outputs:

Monitor / Printer

Validation Rules: None

Special Notes:

1. It will be possible to query audited data deletions By Session ID.
2. Utilizes the logical view SCAudDel_LV which joins SCAudDel_BR, with SCSessLog_BR, and SCSSysUser_BR.

Operation Outline:

START: /* Inquire */

```
    While User Wishes to Continue
        Prompt user for Session ID;
        Starting at that point in SCAudDel_LV, Load a Virtual Data Collection Object with all
            records until End-of-File;
        Display the Virtual Data Collection Object;
    End-While;
STOP.
```

Operations Specification continues ...

E34_A: Add Session Log Entry

Operation Biography:

System:	IMS
Subsystem:	System Controls
Operation Name:	SCSessLog_AO
Operation Description:	Facilitates addition of records to the Session Log file.
Operation Category:	Enhancement
Complexity Rank:	6 of 10
Spec. Author:	E. Foster
Date:	7-10-2006

Inputs:

System Login-in Screen
SCSessLog_BR — Sessions Log (E34)
SCSysUser_BR — System User (E33)

Outputs:

SCSessLog_BR — Sessions Log (E34)
SCSessMark_BR — Sessions Mark (E35)

Validation Rules:

1. Session ID must not previously exist
2. User Code must already exist in SCSSysUser_BR

Special Notes:

1. A Session record is added when a user successfully logs on.
2. The table SCSessMark is accessed and its record incremented with each successful log on.

Operation Outline:

START

```
Set ExitFlag off;
While not ExitFlag
    Display Login prompt for User_Name & Password;
    Check SCSSysUser_BR to determine if valid;
    If valid User
        Retrieve CurrentTime and CurrentDate;
        Use CurrentDate to access SCSessMark_BR and increment its SCSSessionMark by 1;

        // Set Session attributes and write SessionLog
        SCSSessLogID := SCSessMark_BR.SCSessionDate concatenated to
            SCSessMark_BR.SCSessionMark;
        SCSSessLogUser := SCSSysUser_BR.SCUserCode;
        SCSSessLoginTime := CurrentTime;
        Write new record in SCSessLog_BR;
        Set ExitFlag;
        Call SCMainmenu_XO; // The main menu for the system
    Else
        Display Error Message ('Invalid Login Name or Password');
    End-While
STOP.
```

E34_Z: Delete Sessions from Sessions Log

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: **SCSessLog_ZO**
Operation Description: Facilitates deletion of records from the Sessions Log.
Operation Category: Enhancement
Complexity Rank: 6 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs:

SCSessLog_BR — Sessions Log (E34)

Outputs:

SCSessLog_BR — Sessions Log (E34)

Validation Rules: None

Special Notes:

The user will specify a starting Session ID, and an ending Session ID. Records between both sessions (inclusive) will be deleted.

Operation Outline:**START:**

Prompt user for Starting Session and Ending Session;
Delete all records from **SCSessLog_BR** that satisfy the condition
(Start Session <= SCSessLogID <= Stop Session);

STOP

■ **Note** Only authorized personnel must have access to this operation.

E34_I / E34_R: Inquiry/Report on Sessions

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCSessLog_IO / SCSessLog_RO
Operation Description: Facilitates inquiry/report on Sessions.
Operation Category: Enhancement
Complexity Rank: 9 of 10
Spec. Author: E. Foster
Date: 8-10-2006

Inputs:

SCSessLog_BR — Sessions Log (E34)
SCSysUser_BR — System Users (E33)

Outputs:

Monitor / Printer

Validation Rules: None**Special Notes:**

1. It will be possible to query session log via any of the following access paths:
 - 1.1 By Session ID and User Code
 - 1.2 By User Full Name or By User Short Name
2. Each option will invoke one of two sub-operations (SCSessLog_I1 & SCSessLog_I2).
3. Utilizes the logical view SCSessLog_LV, which joins SCSessLog_BR with SCSysUser_BR

Operation Outline:

```
START: /* Inquire */  
      While User Wishes to Continue  
          Present the User with the options mentioned above;  
          Depending on the User's choice, Invoke sub-operation SCSessLog_I1 or SCSessLog_I2;  
      End-While;  
STOP.
```

Outline for SCSessLog_I1:

```
START  
      While User Wishes to Continue  
          Prompt user for Session ID and User Code;  
          Starting at that point in SCSessLog_LV, Load a Virtual Data Collection Object with all records  
              until End-of-File;  
          Display the Virtual Data Collection Object;  
      End-While;  
STOP.
```

{Other sub-operations will be similar}

E35_X: System Initialization Operation:

Operation Biography:

System: IMS
Subsystem: System Controls
Operation Name: SCInit_XO
Operation Description: Initializes the system each day of usage, and sets the initial Session Mark for that day.
Operation Category: Mandatory
Complexity Rank: 3 of 10
Spec. Author: E. Foster
Date: 7-10-2006

Inputs: SCSessMark_BR — Session Mark (E35)

Outputs: SCSessMark_BR — Session Mark (E35)

Validation Rules: None

Special Notes: None

Operation Outline:

START

```
    Retrieve CurrentDate in form YYYYMMDD;  
    Start := 0;  
    // Starter := CurrentDate + Start;  
    Use CurrentDate to check SCSessMark_BR for a record;  
    If (there is no record in SCSessMark_BR with SCSessionMark >= Start)  
        SCSessionMark := Start;  
        SCSessionDate := CurrentDate;  
        Write a new record to SCSessMark_BR;  
    End-If  
STOP.
```

E0_X: System Main Menu Operation

Operation Biography:

System:	IMS
Subsystem:	System Controls
Operation Name:	SCMenu_XO
Operation Description:	Manages the IMS menu and processes user's choices.
Operation Category:	Mandatory
Complexity Rank:	8 of 10
Spec. Author:	E. Foster
Date:	7-10-2006

Inputs: To be determined

Outputs: Monitor

Validation Rules: None

Special Notes:

1. The main menu contains an entry for each of the three subsystems, and a way to exit the system.
2. Each subsystem has its own menu, controlled by sub-operations **SCMenu01_XO**, **SCMenu02_XO**, and **SCMenu03_XO**.
3. The menu items for each menu driver may be hard-coded, or loaded from database tables created for that purpose (hence the undetermined input above).

Operation Outline:

START

While (user wishes to continue)

 Present the main menu;

 If (user does not quit)

 Invoke **SCMenu01_XO**, **SCMenu02_XO**, or **SCMenu03_XO**, depending on the user's choice;

 Else quit;

 End-If;

 End-While;

STOP.

Outline for SCMenu01_XO:

START

While (user wishes to continue)

 Present the Acquisitions Management Subsystem menu;

 If (user does not quit)

 Invoke the operation corresponding to the user's choice;

 Else quit;

 End-If;

 End-While;

STOP.

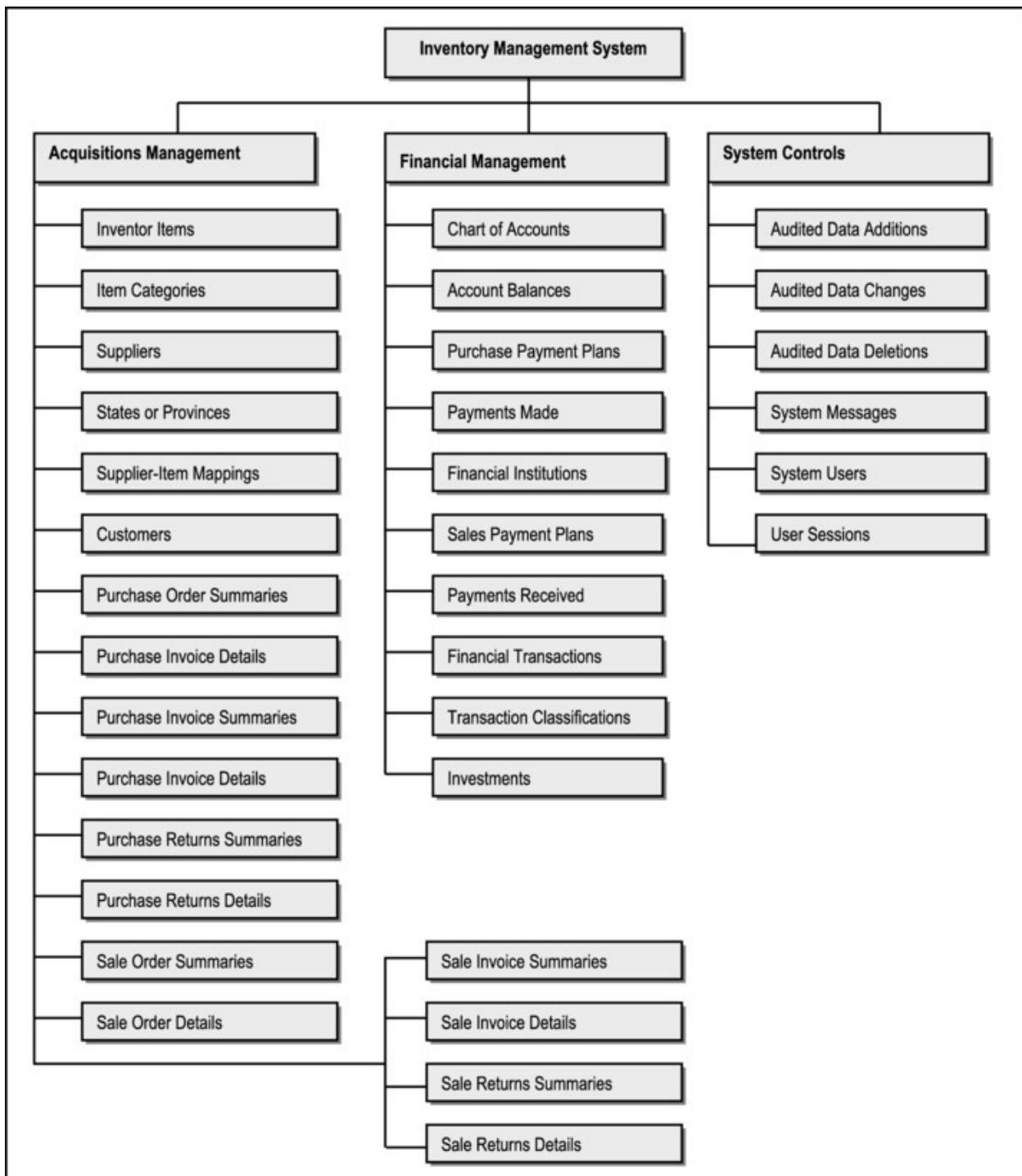
Outline for **SCMenu02_XO** and **SCMenu03_XO** are similar to the outline for **SCMenu01_XO**

A10.4 User Interface Specification

The user interface design is based on Schneiderman's *object-action interface* (OAI) model for user interfaces. The real benefit of this approach is that it is consistent with the way people tend to think: People do not think about the functional intricacies of their daily activities; rather, they think about objects and what they desire to do with them. Because of the natural fit to the typical thought process on the job, user learning will be enhanced.

The menu system will be hierarchical, as represented in the user interface topology chart (UITC) of [Figure A10-10](#). The user interface will be a GUI with the following features:

- At the Inventory management system (IMS):highest level, the main menu will consist of three options representing the three subsystems.
- For each subsystem, the menu options will point to each information entity managed in that subsystem.
- Any option taken from a subsystem menu will invoke a pop-up menu with the operations relevant to that particular information entity (object type).



Inventory Items

Add Inventory Items

Modify Inventory Items

Delete Inventory Items

Inquire / Report on Inventory Items

Item Categories

Add Item Category

Modify Item Category

Delete Item Category

Inquire / Report on Item Category

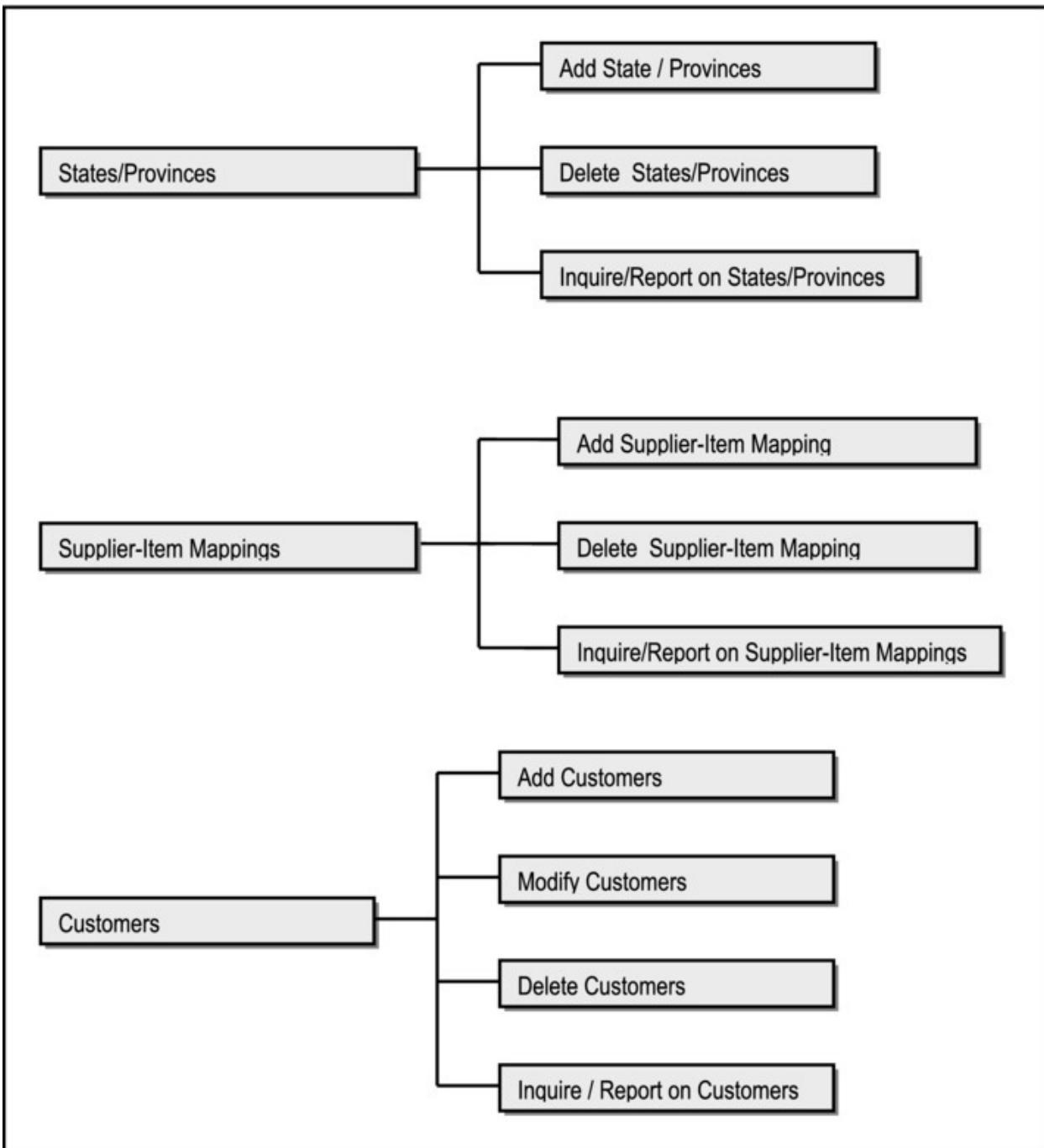
Item Suppliers

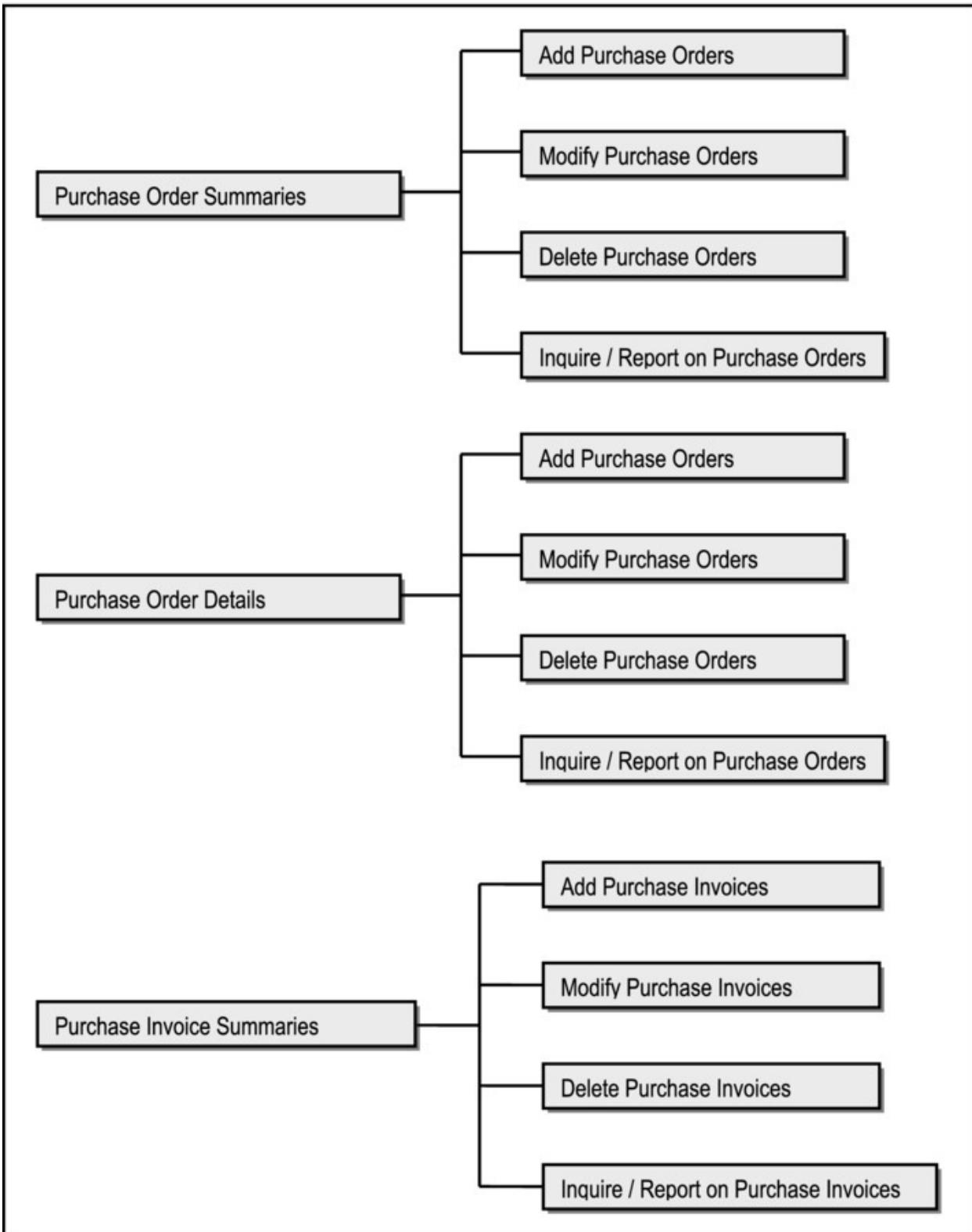
Add Suppliers

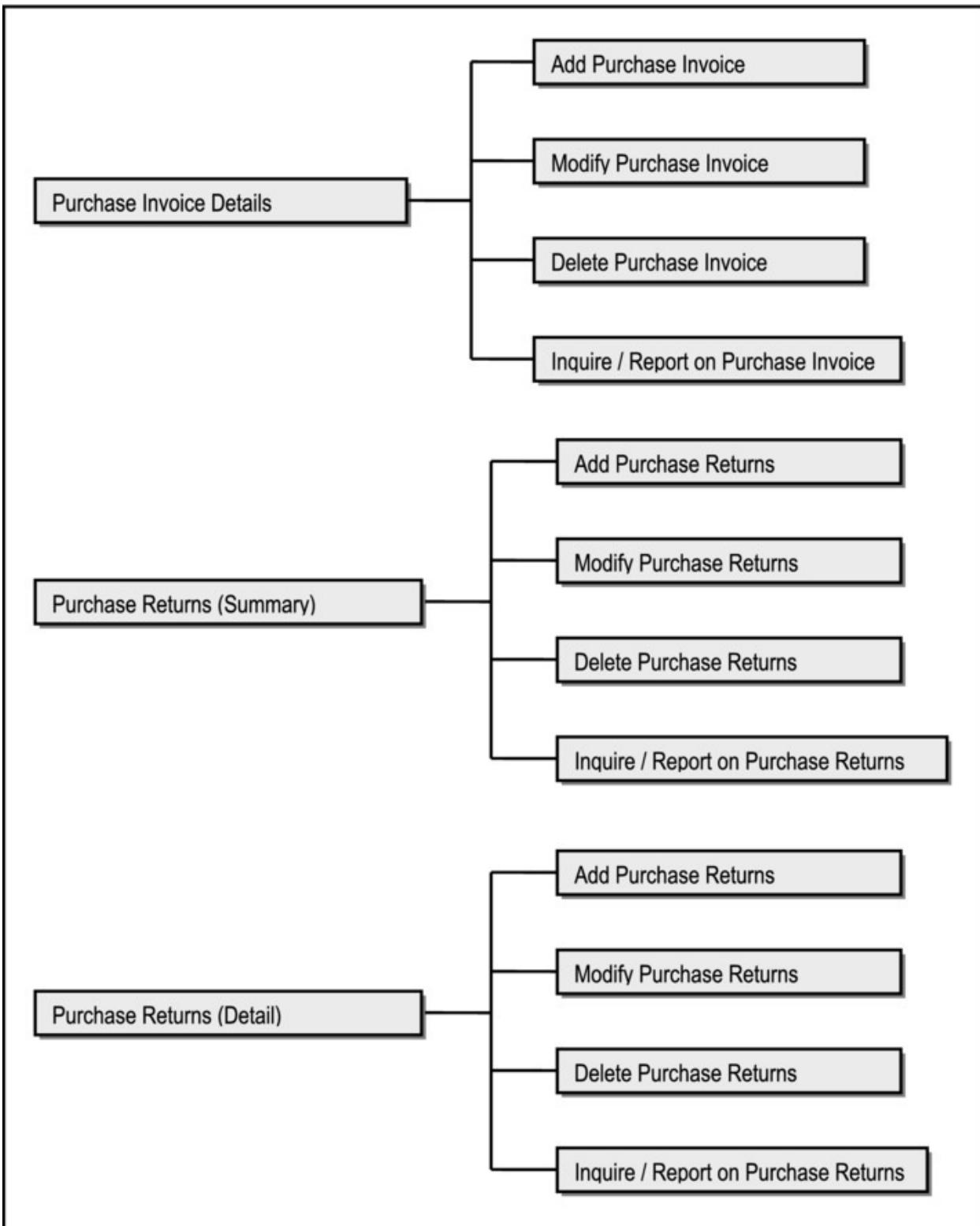
Modify Suppliers

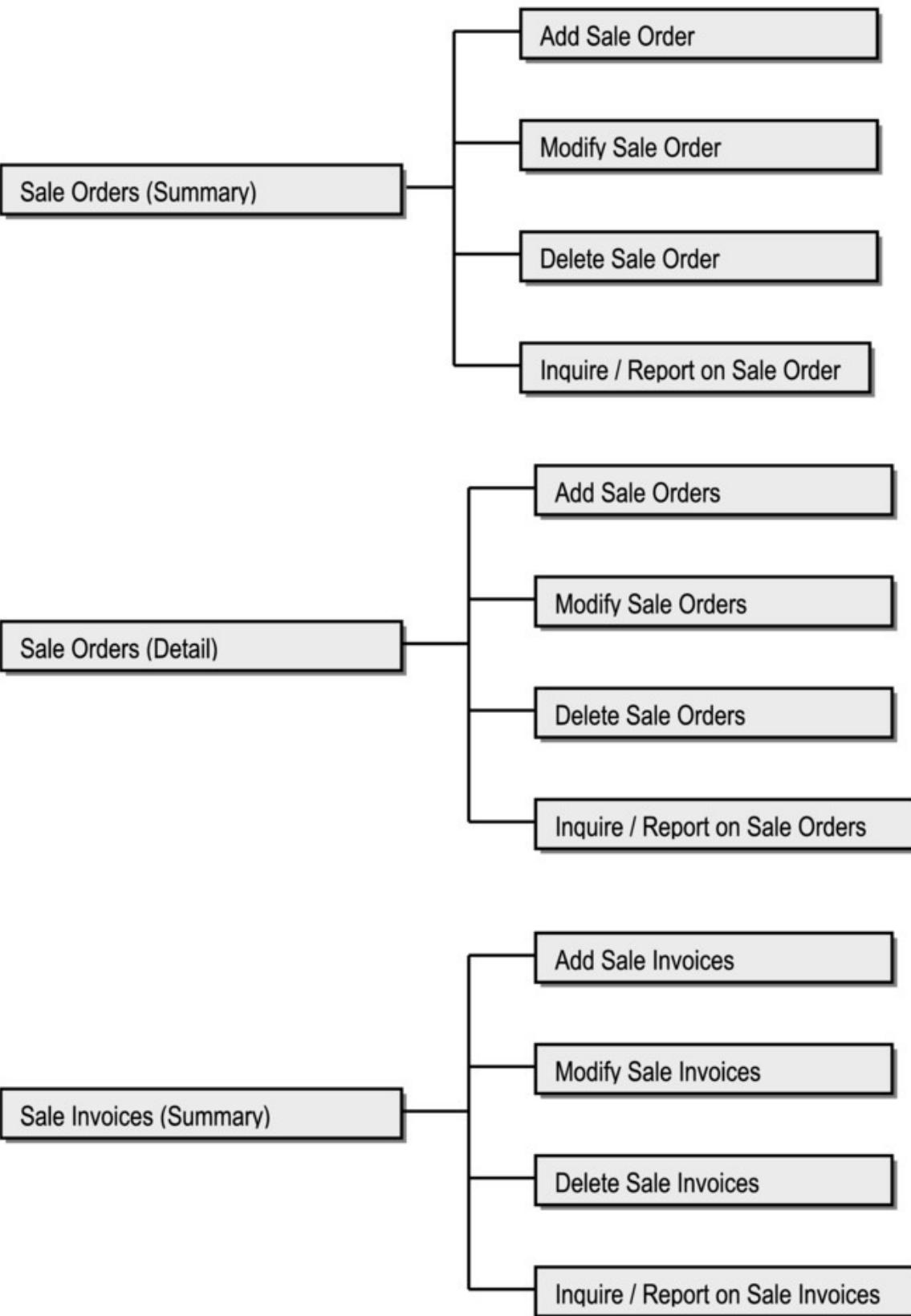
Delete Suppliers

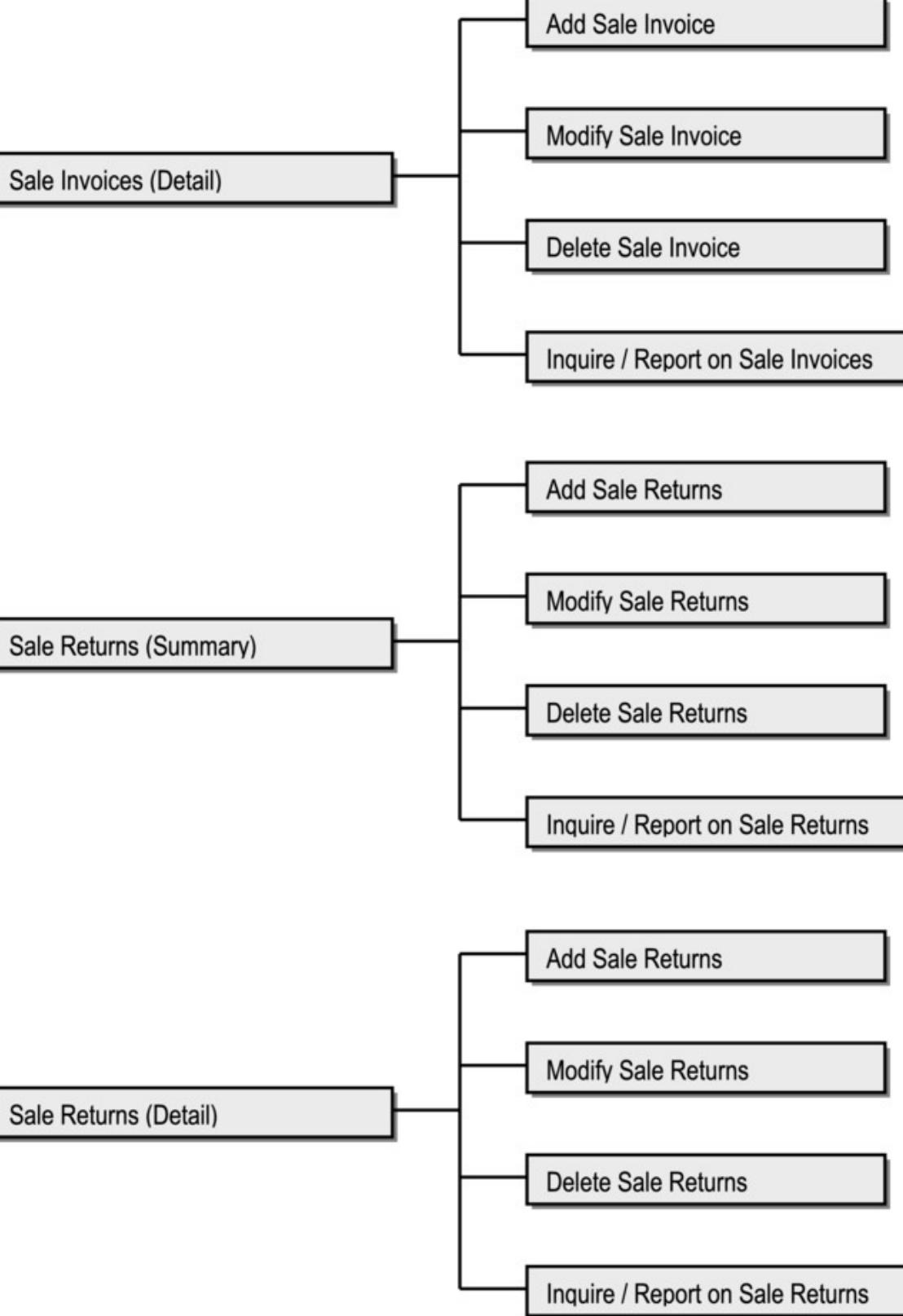
Inquire / Report on Suppliers

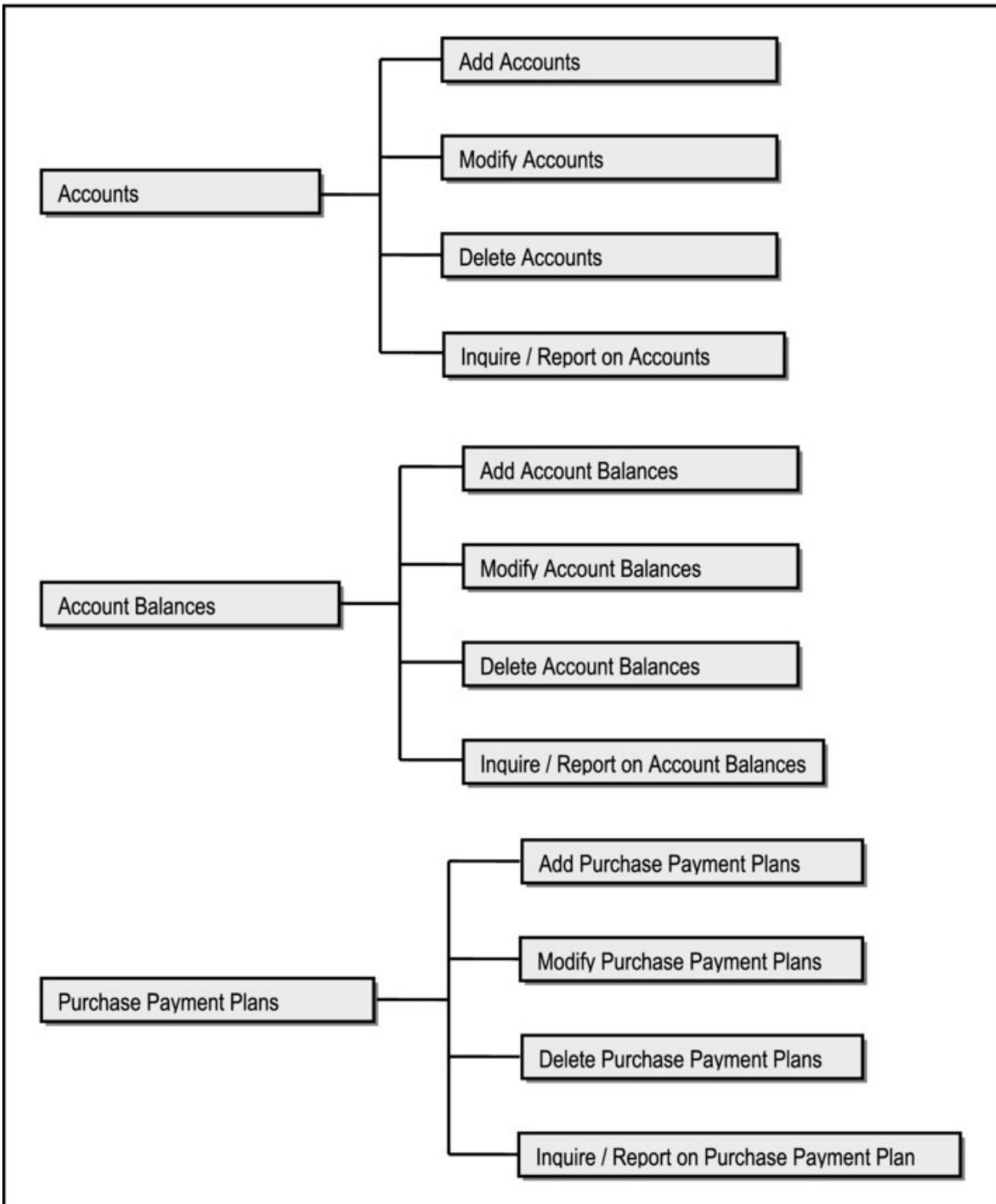


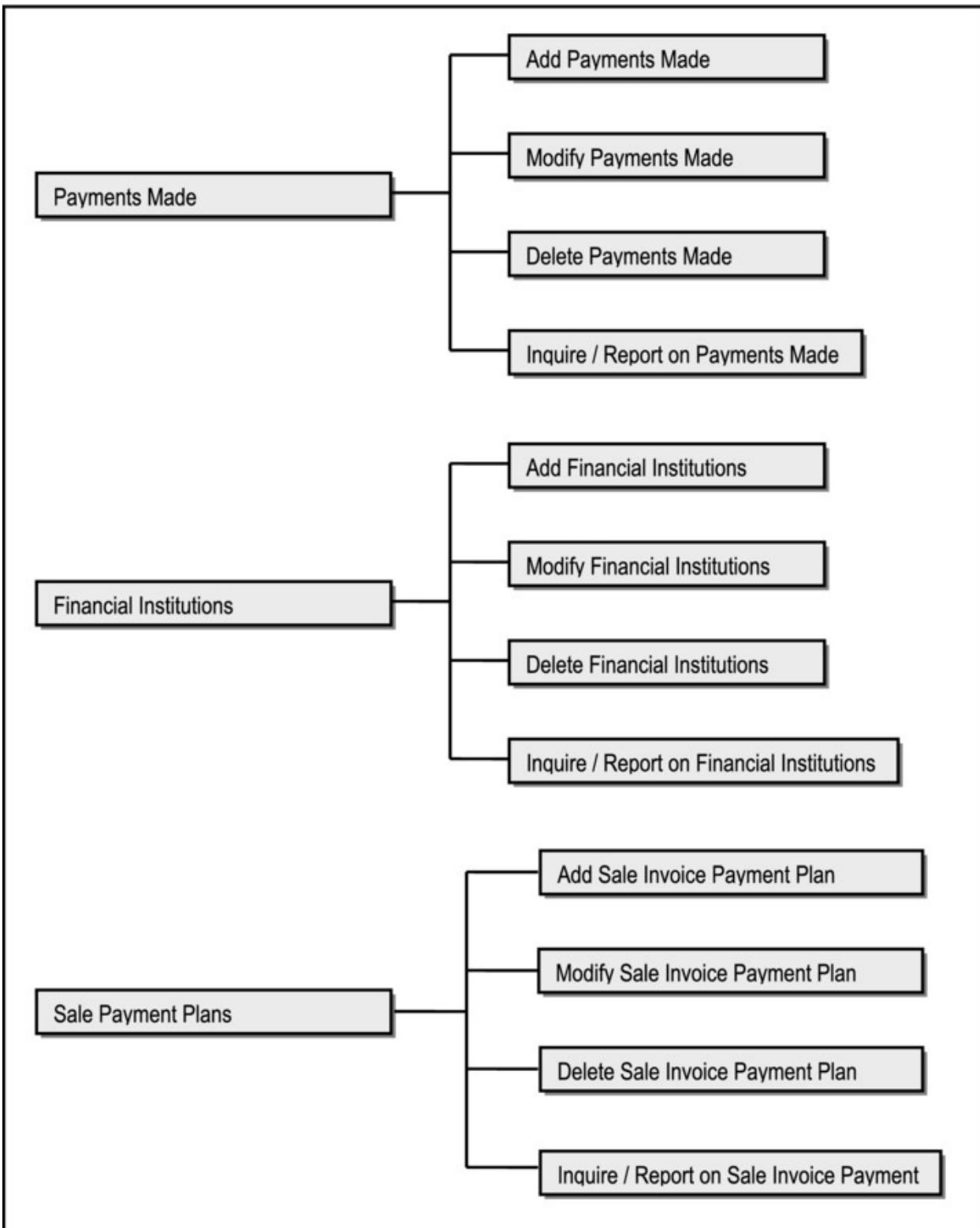


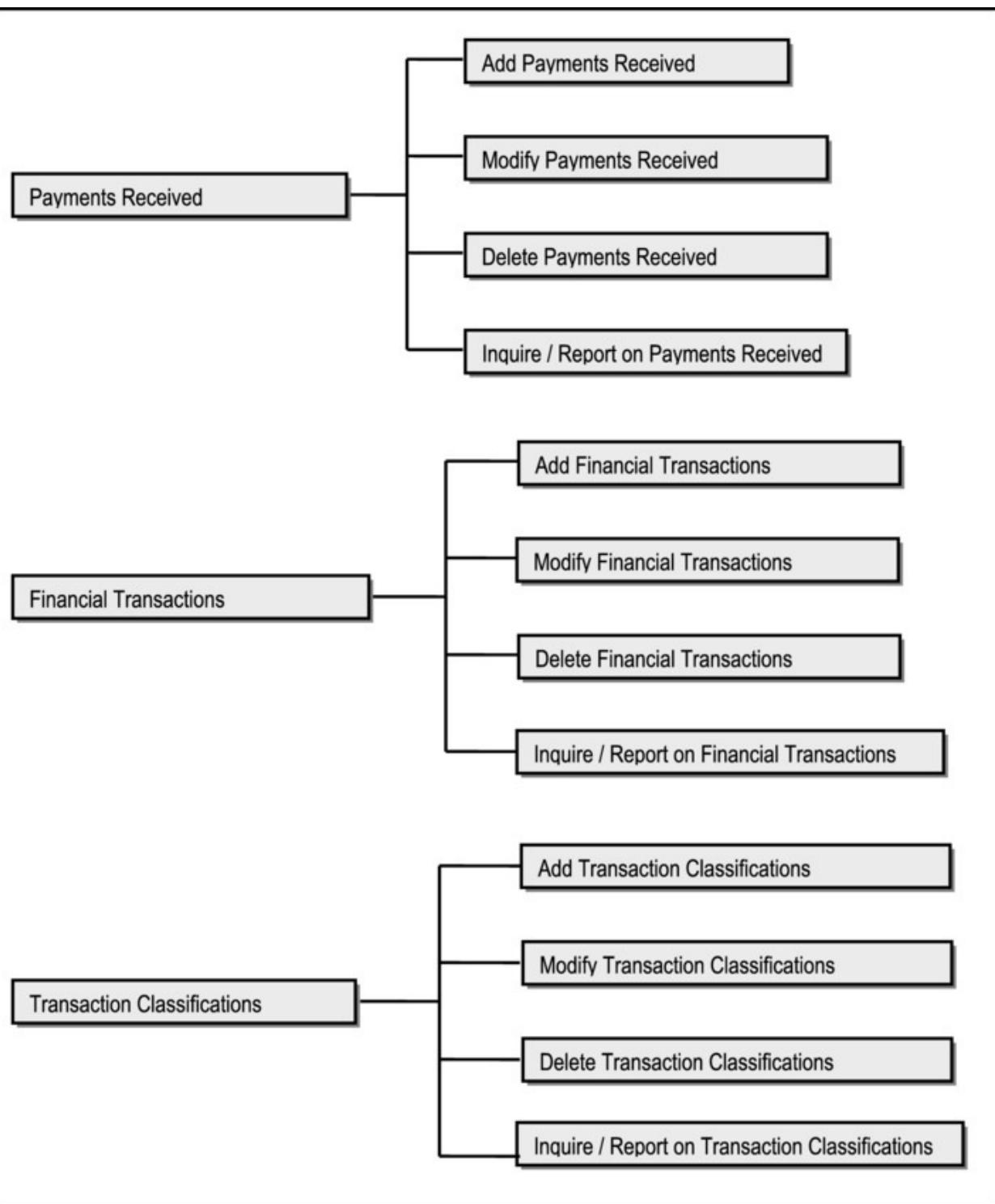


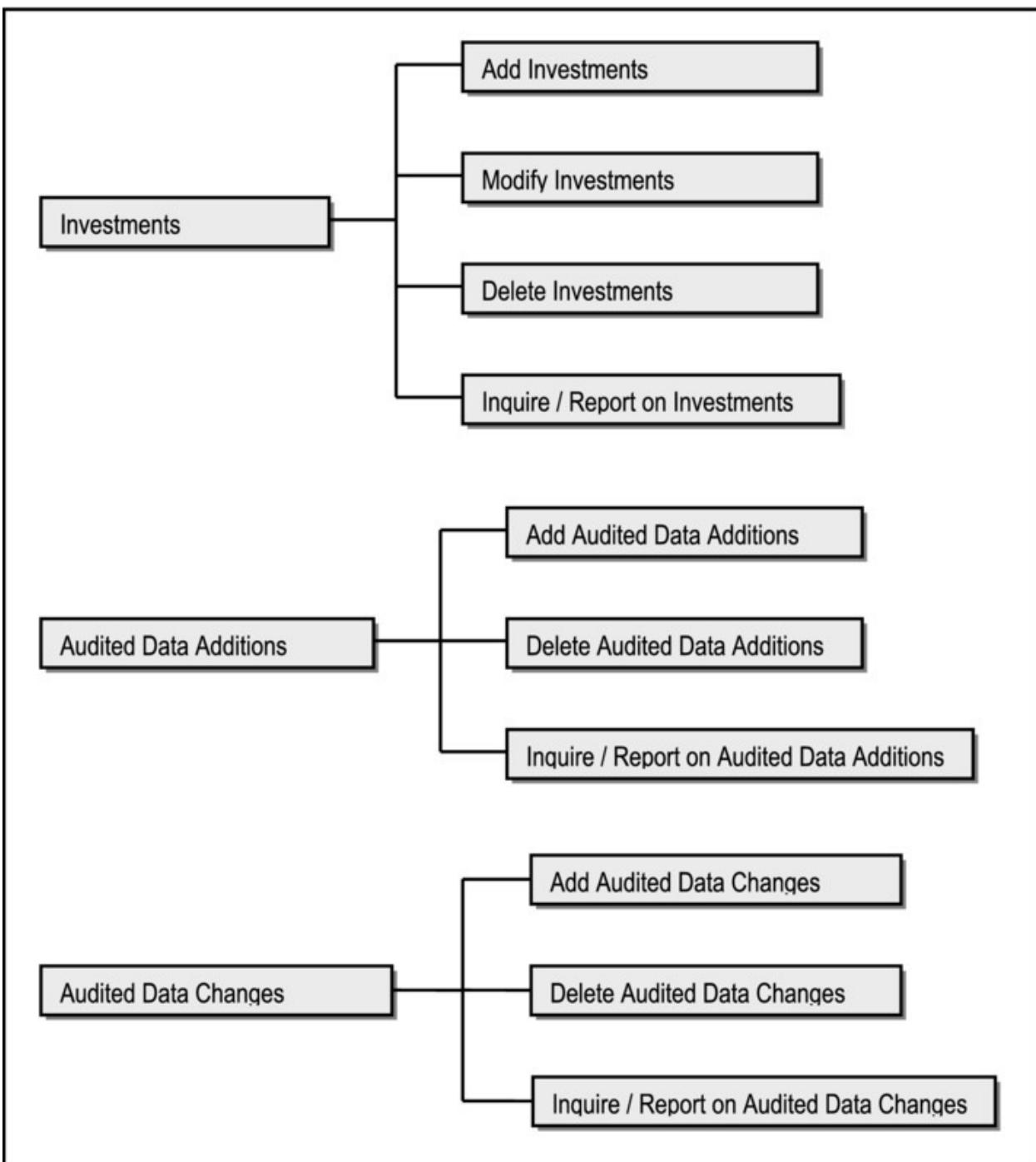












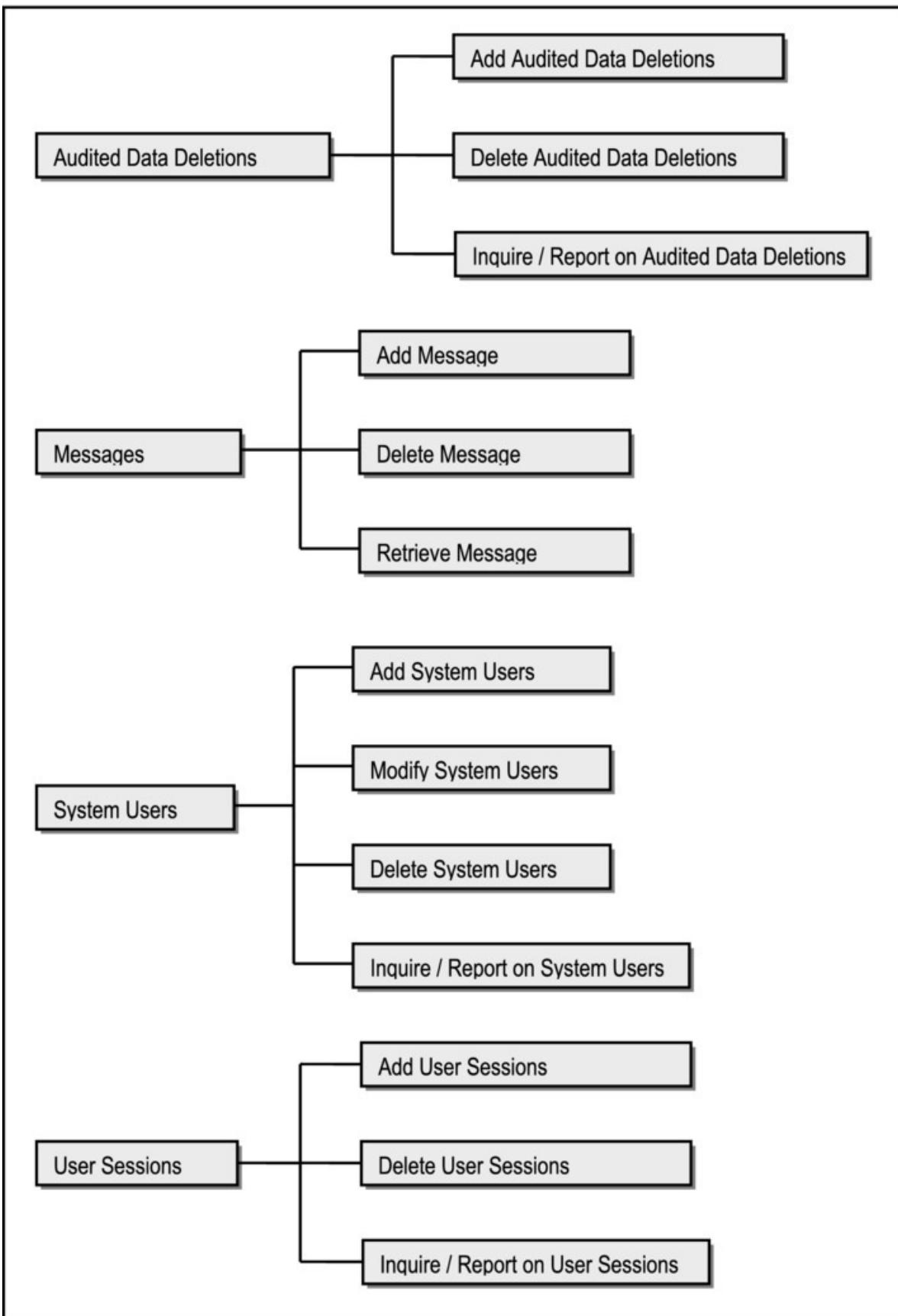


Figure A10-10. UITC for the IMS

This user interface can be easily implemented using an OO RAD tool such as Delphi or NetBeans. Moreover, it can be implemented in one of two ways:

- **Static Approach:** The options may be hard-coded into four menu operations — a main menu and one for each subsystem. For each option taken by the end-user, the appropriate operation would be invoked. This approach is simple but not very flexible. Whenever options are to be changed or new options added, the appropriate menu operation(s) have to be modified.
- **Dynamic Approach:** The options may be loaded into database tables created for that purpose. Each entry would be the option description along with the system name of the actual operation to be invoked. When the user selects an option, the corresponding operation for that option is invoked. The approach provides more flexibility to managers of the system: Menu changes (modifications or option additions) will not necessitate changes to the menu operations. Obviously, this approach requires a bit more intelligent programming effort.

Note Even without OO RAD tools, the user interface prescribed here can still be employed to yield the same benefits.

A10.5 Message and Help Specifications

This brief section addresses message design and help design.

A10.5.1 Message Specification

All user messages will be stored in a system-wide message file (as a database table) called **SCMess_MF** (see entity E32 of [Figure A10-6](#)). Unique codes will be given to each message so they can be accessed easier. Messages will be displayed in the form of pop-up messages.

A10.5.2 Help Specification

The system will host a hypermedia-based help system. Users will access the hypermedia help by clicking appropriate links until they get to the desired help they seek. Additionally, the help system will be organized according to the UITC, so that help will be provided operation-by-operation. As the system matures, the help system can be improved to include context-sensitivity.

A10.6 Summary and Concluding Remarks

This design specification has outlined the blueprint for an Inventory Management System that will be robust, and adaptable to any organization. It includes the following:

- An overview of the system, including problem definition, proposed solution, and system architecture. The overview includes an information topology chart (ITC) that identifies the key information entities in three sub-systems of the system — Acquisitions Management, Financial Management, and System Controls.
- Database specification that includes a set of design conventions, naming conventions, and an O/ESG for each of the information entities comprising the system in all three sub-systems.
- Operations specification that includes an EOS for each operation comprising the system in all three sub-systems.
- User interface specification that outlines menu hierarchy via a user interface topology chart (UITC).

- Message specification that outlines how system messages will be handled, and a help specification that outlines how the help system will be implemented.

Possible future enhancements include the following:

- **Point-of-Sale Subsystem:** This would basically provide a front-end that allows point-of-sale machines to be used. These machines would have to be programmed to interface with the system in a seamless manner.
- **Just-in-Time Subsystem:** This would require automating the re-order process. The idea is to trigger an automatic purchase order to a supplier for goods, once the re-order limit for that item has been reached.
- **Refine the System Controls Subsystem:** More features can be added to the System Controls Subsystem. For instance, a feature could be added that dynamically generates different menus for different users, depending on the features they have been authorized to access.

Despite these potential enhancements, this is a solid start. Now have fun with the development and implementation!

Index

A

Acquisitions management subsystem (AMS)

 inventory items

 addition of

 deletion of

 inquiry/report on

 modification of

 O/ESG

 storage requirements

Acquisitions subsystem (AS)

Activity diagram

Aggregation

Agile development model

Algorithmic cost models

Amalgamation

Amigo Software Inc.

Application composition model

 complexity weights

 engineering effort and complexity

 NOP

 object instance

 object points

 OPC

 productivity rate

Architectural design

 abstract machine model

 availability

 centralized control

 client-server model

 advantages

 challenges

 CUAIS Project

 software systems

 component model

 definition

 event-based control

 maintainability

 performance

 repository model

 advantages

 CUAIS project

 disadvantages

software systems
security
system components

Assignments

ABC Inc.
academic administration system
activity table
BDF inventory management
business community
coding systems
college academic administration system
contemporary software
ERD
HIPO chart
Hotel Management System
IMIS
information topology chart
inputs and information entities
interview/questionnaire
job description
LMX software
management module
methodology
object types
on-line registration
OOD to FOD
operations specification
organization chart
Pay-back
PERT-CPM analysis
POF/DFD
procedure analysis chart
process oriented flow chart
software

- categories
- cost and price
- development tools
- engineering firm
- engineering project
- productivity
- quality
- standards
- system implementation

store manager
Student entity
system life cycle
working environment
Zealot industries

Autocratic management
Automatic validation process

B

Bottom-up approach
Broadcast model
Business process reengineering (BPR)

C

Call-return model
Capability weights (CW)
CASE tools
Class-responsibility-collaboration (CRC) method
Cluster sampling
COCOMO. See Constructive cost model (COCOMO) model
Code-conversion system
Collaboration diagram
College/University Administrative Information System (CUAIS)
Command interface
Component-based software engineering (CBSE)
Component Object Model (COM)
Composition
Computer site preparation
Conflict resolution strategies
Constructive cost model (COCOMO) model
 COCOMO II
 composition model (*see* Application composition model)
 early design model
 post-architecture model
 embedded mode
 organic mode
 semi-detached mode
Contingency leadership
Convenience sampling
CORBA standards
Crashing
Critical path method (CPM)
 advantages
 cost management
 Gantt chart
 planning and preparation
 project activity, scheduling
 project management software
 resource planning and management
 zero slack

D

Database design
 conventional files

ERD

- CASE tool
- Chen notation
- Crow's-foot notation
- manufacturing environment
- OO paradigm
- symbols
- UML notation

file organization

- ISAM
- multi-access file
- relative/direct file
- sequential file

identifying information entities

- O/ESG
- poor database design
- relational/OO database
- relationship identification

- E1 and E2
- manufacturing environment
- testing
- types
- user interface

Database management system (DBMS)

- Database specification (DS)
- acquisitions management subsystem
- financial management subsystem
- object naming conventions
- O/ESG
- systems control subsystem

Data dictionary

- Data flow diagram (DFD)
- advantages
- conventions
- IMS (see Inventory management system (IMS))
- symbols

Decision models

- decision table
 - ambiguity
 - components
 - condition entries
 - redundancy

decision tree

- condition entries
- sale operation

flowcharts

- advantages
- disadvantages

structured language, condition entries

system rules

- action trigger rule
- business rules
- control condition rule
- data integrity rule
- data trigger rule
- declarative vs. procedural statement
- definition
- derivation rule
- OO modeling pyramid
- OO software construction process
- post-condition rule
- pre-condition rule
- relationship integrity rule
- stimulus/response rules
- techniques
 - advantages
 - disadvantages
- Democratic management
- Descriptive narrative approach
- Design of real-time systems. *See* Real-time system design
- Design specification (DS)
 - CASE tools
 - contents
 - definition
 - guidelines
- IMS (*see* Inventory management system (IMS))
- Diagramming techniques
 - activity diagram
 - collaboration diagram
 - DFD (*see* Data flow diagram (DFD))
 - ERD
 - event diagram
 - fern diagram
 - FSM
 - OFD
 - ORD
 - OSD
 - PAC
 - predefined symbols
 - program flowchart
 - advantages
 - control structures
 - symbols
 - software engineering paradigm(s)
 - state transition diagram
 - topology charts (*see* Topology charts)
 - traditional system flowcharts
 - disk/data storage
 - HIPO
 - IOF

module
POF
process/function/operation
subsystem
symbols
use-case diagram
Warnier-Orr diagram

E

Egalitarian management
Encapsulation
Enterprise resource planning (ERP) systems
Entity-relationship diagram (ERD)
Event diagram
Evolutionary development model
Evolutionary prototype. *See* Selected features prototype
Examination
ABC Inc.
Academic Administration System
activity table
Amigo Software Inc.
BDF inventory management
coding systems
components
contemporary software
engineering project
ERD
essential elements
feasibility study
HIPO chart
Hotel Management System
HRMS
inputs and information entities
interview/questionnaire
listing
LMX Software
OOD principles
OOD with FOD
organization
PERT network
POF/DFD
primary functions
process-oriented flow chart
software category
software engineering firm
software life cycle
software standards
store manager
structured decision systems

system evaluation
TIMS
Warnier-Orr diagram
Zealot industries
Extended operation specification (EOS) approach
ADD operation
advantages
algorithms
components
DELETE operation
INQUIRE operation
logical view
object/entity specification grid
UPDATE operation
Virtual Data Collection Object

■ F

Feasibility analysis report
economic feasibility
engineering costs
equipment costs
evaluation matrix
facilities costs
MURRE design factors
operational costs
operational feasibility
PDM strategic factors
quality factors
technical feasibility
TELOS feasibility factors
Fern diagram
Financial management subsystem (FMS)
chart of accounts
account addition
account deletion
account modification
inquiry/report on
O/ESG
storage requirements
Finite state machine (FSM)
First of series prototype
Formal transformation model
Function-oriented design (FOD)
Function-oriented (FO) paradigm

■ G

Generic pseudo-codes
add operation

delete operation
modify operation
Graphical user interface (GUI)

H

Help specification
Hierarchy-input-process-output (HIPO)
High-level language (HLL)
Human-computer interaction (HCI)
Human resource management (HRM)
desired environment
 conflict resolution
 discourages errant actions
 effective communication
 recognizes outstanding achievement
 strategies
 uncomfortable situations
grooming and succession planning
hiring
 checklist
 important considerations
 interview evaluation form
 preparation
preserving accountability
 designing and assigning work
 MBO program
 performance evaluation
 sample employee appraisal
 sample work schedule
 strategies
responsibilities
styles
 autocratic management
 contingency leadership
 democratic management
 laissez faire management
 path-goal leadership
 relation-oriented leadership
 super leader approach
 task-oriented leadership
 transformational leadership
Human resource management system (HRMS)
Hypermedia-based help system

I

Indexed sequential access method (ISAM)
Information engineering CASE (IE-CASE)
Information engineering (IE)

Information gathering
brainstorming
interviewing
mathematical proofs
object identification (see Object identification)
observation and document review
prototyping
questionnaires
 administering
 guidelines
 scales
 situations
requirements analysis
requirements specification (RS)
sampling
 delivery of orders after customer complaints
 non-probability sampling techniques
 probability sampling techniques
 sample calculations

Information-oriented flowchart (IOF)

Information topology chart (ITC)

 CUAIS project
 enterprise planning
 object categorization
 advantages
 component relationships
 definition
 vs. fern diagram
 HIPO chart
 IMS
 limitations

Inheritance

Initial system requirement (ISR)
 feasibility analysis report (see Feasibility analysis report)
 initial project schedule
 problem definition
 employees behavior observation
 external constraints
 guidelines
 internal constraints
 performance criteria
 source documents examination
 user complaint
 vendors external feedback
 work process examination

 project team
 proposed solution
 system justification

Interfaces

Interrupt-driven model

Inventory items
addition of
deletion of
inquiry/report on
modification of
Inventory Management Information System (IMIS)
Inventory management system (IMS)
benefits
business rules
 data integrity rules
 derivation and procedural rules
 relationship integrity rules
components
context diagram
database specification
 acquisitions management subsystem
 financial management subsystem
 object naming conventions
 systems control subsystem
data storage objects
definition
economic feasibility
feasibility evaluation grid
help specification
important processes
information topology chart
initial project schedule
level-1 DFD
level-2 DFD
message specification
object flow diagram
objectives
operational feasibility
operational requirements
operations specification
 acquisitions management subsystem
 FMS (*see* Financial management subsystem (FMS))
 generic pseudo-codes
 SCS (*see* System controls subsystem (SCS))
outputs
portable and platform independent
salient features
storage requirements
 acquisitions management subsystem
 financial management subsystem
system overview
technical feasibility
UITC
user interface specification
user operations list

Iterative development model

J, K

Job descriptions

Joint Application Design (JAD)

Joint Enterprise Modeling (JEM)

Joint Requirements Planning (JRP)

Judgment sampling

Just-in-time subsystem

L

Laissez faire management

Legacy system

Legacy systems

LMX Software Inc.

M

Maintainability, usability, reusability, reliability, extendibility (MURRE) factors

MAINTAIN operations

Management by objectives (MBO)

Management support systems (MSSs)

Manager model

Manual validation process

Martin's information system pyramid

Menu driven interfaces

Message specification

Microfilm/microfiche

Multiple inheritances

Multithreading

N

.NET products

Non-operational/interactive prototype

Non-probability sampling techniques

O

Object-action interface (OAI) model

Object behavior

activity diagrams

collaboration diagrams

event diagrams

control conditions

events trigger operations

function-oriented approach

object type

post-conditions

- pre-conditions
- purchase order process
- state transition diagram
- subtypes and super-types

FSM

OFD

- sequence diagrams
- states and state transition
- triggers

use-case

- actors
- common entities
- concrete operations
- convey information
- definition
- essential operations
- high-level operation
- low-level operation
- primary operations
- registration process
- same actor, same state
- secondary operations
- specific workflow
- symbols

Object behavior analysis (OBA)

Object categorization

- bottom-up approach

CRC card

fern diagram

- advantages
- drawbacks
- instances
- network structured
- tree structured

identify relationships

ITC

- advantages
- component relationships
- definition
- vs. fern diagram
- HIPO chart
- IMS
- limitations

multiple inheritance

- delegation and inheritance
- delegation via aggregation
- interfaces
- nested generalization
- workarounds

ORD

- abstract types
- depicting inheritance, college community
- ERD
- OO-CASE tool
- OO-GUI
- star schema
- OSD
- top-down approach
- Object/entity specification grid (O/ESG)
 - acquisitions management subsystem
 - financial management subsystem
 - systems control subsystem
- Object flow diagram (OFD)
- Object identification
 - descriptive narrative approach
 - information entity
 - object type
 - rule-of-thumb approach
 - techniques
- Object oriented analysis and design (OOAD)
- Object oriented-CASE tools
- Object oriented design (OOD)
 - advantages
 - amalgamation
 - inheritance
 - interaction
 - ITC
 - object type
 - observation
 - polymorphism
 - strategies
 - tools
- Object oriented domain analysis (OODA)
- Object oriented enterprise modeling (OOEM)
- Object oriented-ICASE tools
- Object-oriented information engineering (OOIE)
 - BAA
 - object/database structure
 - object-operation matrices
 - object structure diagrams
 - O/ESG
 - security mechanism
 - system-component-business-area matrices
 - BPR
 - characteristics
 - diagramming techniques
 - enterprise planning
 - activities
 - ITC
 - ODF

UITC
infrastructure
objectives
OOEM
system construction
system design
Object-oriented methodologies (OOM)
amalgamation
benefits
CASE tools
 automatic code generation
 benefits
 categories of
 classifications
 fragmented CASE
 GUI
 information engineering CASE (IE-CASE)
 instant CASE
 integrated CASE (I-CASE)
 repository stores
characteristics
class
encapsulation
end user involvement
enterprise-wide design
information engineering
inheritance
intangible/conceptual objects
interfaces
late binding
methods
modeling and code generation
multithreading
object identification
 applicable techniques
 CRC method
 decomposition
 descriptive narrative approach
 generalizations and subclasses
 hierarchies, individual objects and classes
 objects, categories and interfaces definition
 OODA/application framework
 personal experience
 rule-of-thumb method
 skilled analysis
 things to be modeled
object type
OOAD
OO-CASE tools vs. OO-programming language
OO diagramming

- OO modeling
- OOPL
 - operations
 - perception vs. reality
 - polymorphism
 - requests
 - reusability
- SDLC
 - advantages
 - function-oriented approach
 - object oriented approach
- software revolution
- standards for
 - COM
 - CORBA
 - .NET Boom
- tangible objects
- UDBMS
- Object-oriented (OO) paradigm
- Object oriented programming languages (OOPLs)
- Object-oriented software engineering (OOSE)
- Object-oriented technology (OOT/OT)
- Object point count (OPC)
- Object-relationship diagram (ORD)
 - abstract types
 - depicting inheritance, college community
 - ERD
 - OO-CASE tool
 - OO-GUI
 - star schema
- Object structure analysis (OSA)
- Object structure diagram (OSD)
- Office preparation
- Operational requirements
- Operations design
 - categorization
 - development environment
 - formal specifications
 - hybrid environment
 - informal methods
 - EOS approach (*see* Extended operation specification (EOS) approach)
 - UML notations (*see* Unified modeling language (UML))
 - Warnier-Orr diagram
- OOD paradigm
- unique system name
- Operations specification
 - AMS (*see* Acquisitions management subsystem (AMS))
 - generic pseudo-codes
 - add operation
 - delete operation

modify operation
SCS (see System controls subsystem (SCS))

Organization

definition
functional areas
information levels
junior management
middle management
open source community
operational staff
organizational chart
top management

Organizational structure

factors
functional approach
advantages
disadvantages
medium/small organization
hybrid/matrix organization
parallel/project-oriented organization
software engineering firms
organization chart
product lines

Organizational theory and behavior (OTB)

P

Patched-up prototype
Path-goal leadership
PERT/CPM
Phased prototype model
Point of sale subsystem (POSS)
Printed output
Probability sampling techniques
Procedure analysis chart (PAC)
Process-oriented flowchart (POF)
Product documentation
system help facility
content
structure
user's guide and system guide
Production model
Productivity, differentiation and management (PDM) factors
Program description languages (PDLs)
Program evaluation and review technique (PERT)
activity event
advantages
diagram
Gantt chart
planning and preparation

- project activity, scheduling
- project management software
- sensitivity analysis

Program flowchart

- advantages
- control structures
- symbols

Project selection

■ Q

Quality assurance (QA)

Questionnaires

- administering
- guidelines
- scales
- situations

Quota sampling

■ R

Rapid prototype model

Rational Unified Process (RUP)

Real-time executive

Real-time system design

- modeling
- programming
- stages

Relational databases

Relation-oriented leadership

Requirements analysis

Requirements specification (RS)

- acknowledgments
- appendices
- concluding remarks
- definition
- detailed requirements
- documenting

IMS (*see* Inventory management system (IMS))

interface specification

preparation steps

presentation

problem synopsis

refined project schedule

system components

system constraints

system documentation

system overview

system security requirements

validation process

Rule-of-thumb approach

Rule-of-thumb method

S

SCMess_MF

Screen output

Selected features prototype

Simple random sampling

Snowball sampling

Software, definition

Software design

 architectural design (*see* Architectural design)

 development standards

 advantages

 issues

 documentation

 FOD

 interface design

 message design

 object structure

 OOD

 advantages

 amalgamation

 inheritance

 interaction

 ITC

 object type

 observation

 polymorphism

 strategies

 tools

 operations

 security design

 specification

 CASE tools

 contents

 definition

 guidelines

 UML

 user interface

Software development

Software development issues

 design specification

 implementation strategy

 leadership and motivation

 life cycle strategies

 standards and quality assurance

 QA evaluation

 relationship between quality and standards

- software quality factors
- targets and financial resources
 - budget and expenditure
 - software cost and value
- Software development life cycle (SDLC)
 - advantages
 - function-oriented approach
 - object oriented approach
 - phases
- Software economics
 - cost components
 - cost estimation
 - algorithmic cost models
 - analogy
 - COCOMO (*see* Constructive cost model (COCOMO))
 - consumer pricing
 - expert judgment
 - Parkinson's law
- price
- productivity
 - function-related metrics
 - size-related metrics
 - value-added assessment
- value
- Software engineer
 - CASE tools
 - coding systems
 - advantages
 - alphabetic codes
 - block sequence
 - digit
 - features
 - group classification
 - self-checking code
 - simple sequence
 - data analysis charts
 - DBMS suites
 - forms design
 - advantages
 - body
 - cut sheet forms
 - electronic forms
 - fill-in styles
 - footing
 - guiding principles
 - heading
 - specialty forms
 - historical role
 - job description
 - management issues

- modern role
- software planning and development tools
- technical documents
- Software engineering
 - acquisition approaches
 - analysis process
 - categories
 - firms, product lines
 - function-oriented (FO) approach
 - objective
 - object-oriented (OO) approach
 - system transformation path
- Software implementation issues
 - code conversion
 - direct changeover
 - distributed conversion
 - marketing strategy
 - operating environment
 - centralized system
 - distributed system
 - parallel conversion
 - phased conversion
 - system installation
 - training strategy
- Software integration
- Software life cycle
 - agile development model
 - component-based approach
 - formal transformation model
 - iterative development model
 - phased prototype model
 - rapid prototype model
 - SDLC phases
 - waterfall model
- Software management
 - factors
 - integration
 - legacy system
 - maintenance
 - cost
 - modifications
 - patch
 - upgrade
 - re-engineering
- Software productivity
 - function-related metrics
 - size-related metrics
 - value-added assessment
- Software quality factors
- Software reuse

State-diagram. *See* Finite state machine (FSM)
State transition diagram
Static methods
Storage requirements
 acquisitions management subsystem
 financial management subsystem
Stratified sampling
Super leader approach
Systematic random sampling
System catalog
 building
 business rules
 database-related contents
 illustration
 operational contents
 system documentation
System controls subsystem (SCS)
 audit table
 data additions
 data deletions
 data modifications
O/ESG
Session Log
 addition of
 deletion of
 inquiry/report on
system initialization operation
system main menu operation
System, definition
System initialization operation
System main menu operation
System security
 access to system
 access to system data
 access to system resources

■ T

Task-oriented leadership
Throw-away prototype
Top-down approach
Topology charts
 ITC
 UITC
Total quality management (TQM)
Training Information Management System (TIMS)
Transformational leadership

■ U, V

Unified modeling language (UML)

activity diagram

collaboration diagram

state diagram

state diagrams

Use-case diagram

Universal database management system (UDBMS)

Use-case diagram. *See also* Data flow diagram (DFD)

Use cases

actors

common entities

concrete operations

convey information

definition

essential operations

high-level operation

low-level operation

primary operations

registration process

same actor, same state

secondary operations

specific workflow

symbols

User interface design

command interface

considerations

GUI

HCI

human factors

input

guidelines

objectives

interaction styles

menu driven interfaces

output

audio output

guidelines

magnetic storage devices

method selection

microfilm/microfiche

monitor display

objectives

optical storage devices

printed output

preparation

user needs

User interface specification

dynamic approach

static approach

UITC

User interface topology chart (UITC)

User message management

error messages

message retrieval

status messages

storage and management

warning messages

■ **W, X, Y, Z**

Warnier-Orr approach

advantages

disadvantages

for MAINTAIN operation

Warnier-Orr diagram

Waterfall model

Working model